Framework eMobc IOS

Manual del Usuario

Versión 0.1

AVISO LEGAL

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. **Definitions**

a."**Adaptation**" means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

b."**Collection**" means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(g) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.

c."**Distribute**" means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.

d."**License Elements**" means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, Noncommercial, ShareAlike.

e."**Licensor**" means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

f."**Original Author**" means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

g."**Work**" means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a

compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

h."**You**" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

i."**Publicly Perform**" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

j."**Reproduce**" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. **Fair Dealing Rights**. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. **License Grant**. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a.to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

b.to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

c.to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

d.to Distribute and Publicly Perform Adaptations.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights described in Section 4(e).

4. **Restrictions**. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a.You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as

required by Section 4(d), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(d), as requested.

b.You may Distribute or Publicly Perform an Adaptation only under: (i) the terms of this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-NonCommercial-ShareAlike 3.0 US) ("Applicable License"). You must include a copy of, or the URI, for Applicable License with every copy of each Adaptation You Distribute or Publicly Perform. You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License. You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.

c.You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in con-nection with the exchange of copyrighted works.

d.If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and, (iv) consistent with Section 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(d) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

e.For the avoidance of doubt:

i.**Non-waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

ii.**Waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License if Your exercise of such rights is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(c) and otherwise waives the right to collect royalties through any statutory or compulsory licensing scheme; and,

iii.**Voluntary License Schemes**. The Licensor reserves the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers

voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License that is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(c).

f.Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. **Representations, Warranties and Disclaimer**

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING AND TO THE FULLEST EXTENT PERMITTED BY APPLICABLE LAW, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THIS EXCLUSION MAY NOT APPLY TO YOU.

6.**Limitation on Liability**. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. **Termination**

a.This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b.Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. **Miscellaneous**

a.Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b.Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

c.If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

d.No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e.This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

f.The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

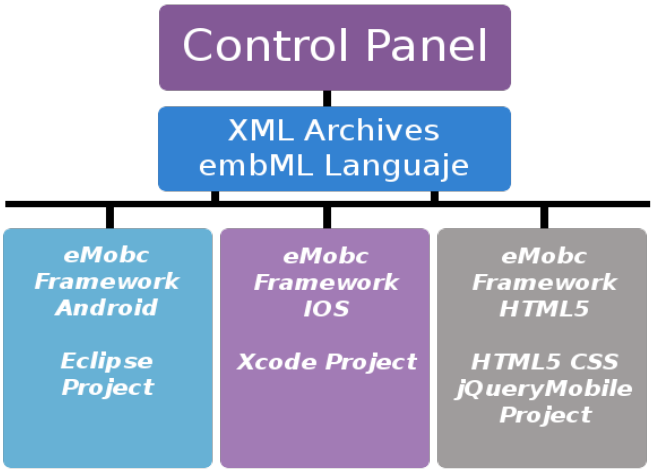
Índice de contenidos

Página en blanco – separador

Descripción general

Emobc es un framework que permite generar aplicaciones móviles definidas desde un panel de control para distintas plataformas: Android, iOS y HTML5.

El framework está basado en archivos xml. Una vez se ha definido la aplicación que se quiere crear desde el panel de control, éste genera una serie de archivos xml que describen todas las propiedades, ventanas, y funciones que tendrá la aplicación.



Cada plataforma dispone de un proyecto base donde se almacenarán estos archivos xml. Durante la ejecución de la aplicación, son parseados a medida que son necesarios para ir generando la aplicación a partir de la información recopilada.

El código necesario para ejecutar la aplicación ya está disponible en los proyectos base. Tan solo es necesario añadir los archivos xml y compilar el proyecto para obtener una aplicación funcional.

App.xml

El archivo xml principal es el app.xml (Ver aplication data). Toda la información general que se aplica a la aplicación está en este archivo como el tipo de publicidad, las rotaciones soportadas, los menús de navegación (ver menús) o el Entry Point (ver Entry Point).

El EntryPoint básicamente es la pantalla que se mostrará en primer lugar (por defecto será la portada).

Levels y NextLevels

Los XML definen tanto la aplicación en general (app.xml) como cada una de sus pantallas. Estas pantallas están estructuras en el framework como levels. Un level es una ventana de la aplicación. Es de un tipo concreto y tiene unos datos asociados.

Todos los levels que contiene una aplicación estarán definidos en su app.xml.

Las acciones dentro de las pantallas de la aplicación, como pulsar un botón o una imagen, pueden tener asociados un nextLevel. Un nextLevel es la próxima ventana que se mostrará cuando el usuario realice la acción a la que está asociado. (ver Levels)

```
<levels>
        <!-- web -->
        <level>
        <levelId>web</levelId>
        <levelTitle>Ventana Web</levelTitle>
        <levelFile>web.xml</levelFile>
        <levelType>WEB_ACTIVITY</levelType>
        </level>
```

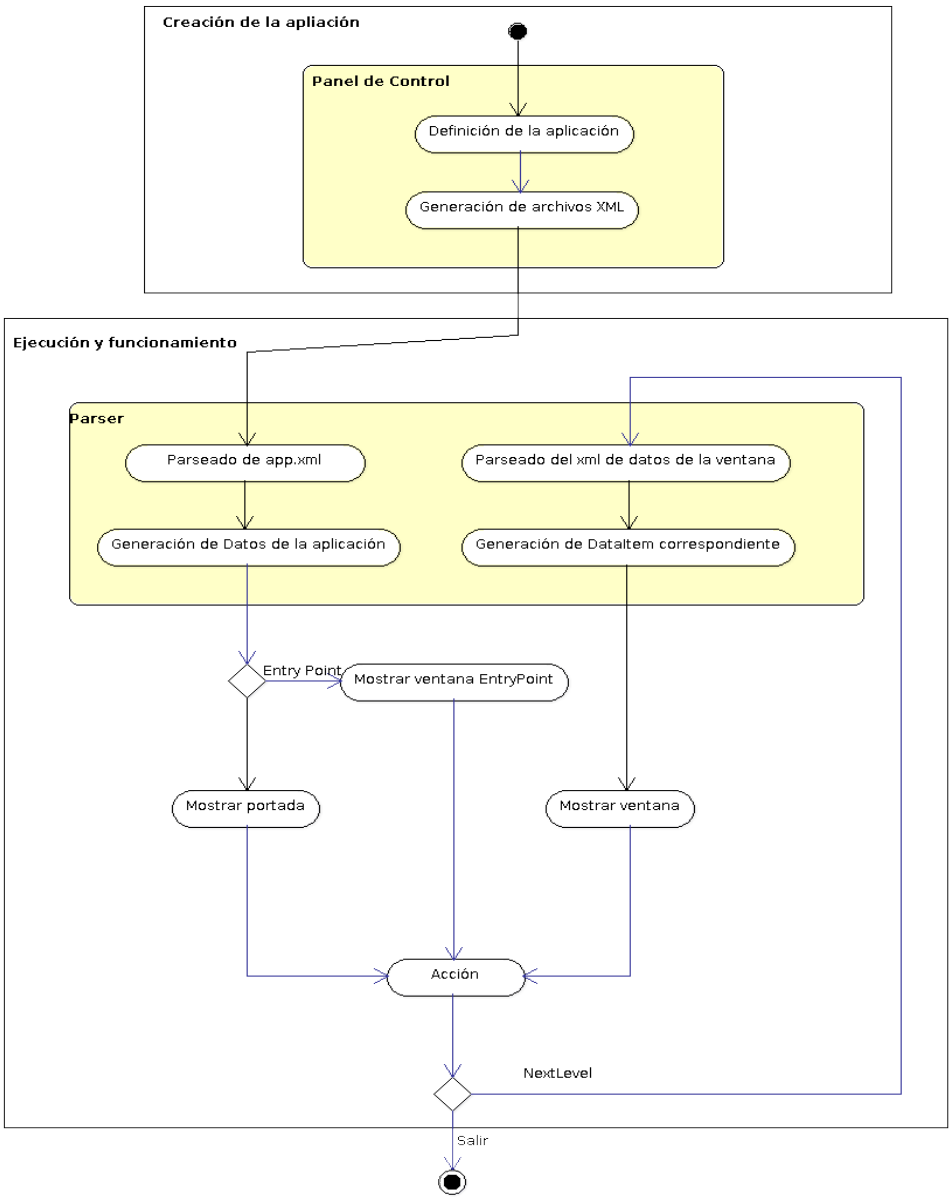
Funcionamiento de la aplicación

El funcionamiento de una aplicación construida con el framework se basa principalmente en parseado de los archivos XML que definen la aplicación. El primer archivo que se parsea es app.xml. Se generarán todos los datos de la aplicación.

Una vez la portada o el entry point sea visible, cada acción del usuario puede llevar a nuevas ventanas (nextLevel) o a salir de la aplicación.

Cuando se realiza una acción que tiene asociado un next level, la aplicación busca el archivo de datos xml asociado a ese nextLevel y lo parsea para recuperar la información y crear la pantalla que se va a mostrar. (Ver levels).

Conviene diferenciar, tanto en Android como en iOS, la creación de la aplicación desde el panel de control del funcionamiento. El panel de control es el que genera los archivos xml y el proyecto en Android e iOS es el que va construyendo, a partir de esos xml, la aplicación en tiempo de ejecución.



Level

Estructura de las aplicaciones

Las aplicaciones que genera el framework se estrucutran en base a levels (niveles). Un level es un tipo de ventana que tiene la aplicación. Por esa razón se incluyen todos los levels de una aplicación en el app.xml.

Next Level

Cuando en una ventana tiene un botón, y al pulsarlo tiene asociado un enlace a una nueva ventana, decimos que tiene un nextLevel asociado a la acción de pulsar el botón.

Un nextLevel es el level que se creará cuando se realiza una acción. Para definir un nextLevel es necesario tanto el level que queremos crear, como los datos asociados a él que la aplicación tendrá que cargar.

```
Portada.xml
<button>
        <buttonTitle>Navegador</buttonTitle>

<buttonFileName>images/buttonw.png</buttonFil
eName>
        <nextLevel>

<nextLevelLevelId>web</nextLevelLevelId>

<nextLevelDataId>web1</nextLevelDataId>
        </nextLevel>
</button>
```

Diferencia entre level y ventana

Un level no es una de todas las pantallas que tendrá la aplicación. Si una aplicación tiene tres tipos de ventana diferentes (por ejemplo: calendario, lista con imágenes y lector pdf) tendrá tres levels definidos en su xml. Sin embargo, otra aplicación puede tener en total 3 ventanas, todas ellas de tipo contenedor web y tan solo tendrá un level declarado en su app.xml.

Para diferenciar cada una de estas 3 ventanas entre sí, porque previsiblemente querremos que contengan diferente información, se utilizan los datos asociados al level, definidos mediante el tag “nextLevelDataId” de la llamada al nextLevel (ver figura 1).

```
<levels>
        <!-- web -->
        <level>
        <levelId>web</levelId>
        <levelTitle>Ventana Web</levelTitle>
        <levelFile>web.xml</levelFile>
        <levelType>WEB_ACTIVITY</levelType>
        </level>
```

Asociación de datos de los levels

Para identificar cada level y sus datos se utilizan identificadores de tipo String. Un level tiene un identificador único y un archivo xml de datos asociado. Cada vez que se quiere crear un nextLevel se utiliza el identificador del level (que está definido en el app.xml) y el identificador de los datos.

Cada vez que se llama a un nextLevel desde la aplicación, se consulta la información del app.xml por levelId y se leen los datos identificados por el tag “nextLevelDataId” dentro del archivo xml especificado en el tag ”levelFile”.

Un archivo de datos xml puede tener varios datos dentro. Cada uno tiene un identificador único al que se puede hacer referencia en cada llamada a un nextLevel.

A continuación se muestra un ejemplo para clarificar el funcionamiento de los levels y los datos asociados. Continuando con el escenario anterior, en el que se tienen tres ventanas de tipo contenedor web, vamos a ver cómo se referencian entre sí los tags y los archivos xml.

Primero se presentarán cómo son nuestros archivos xml y qué significa cada tag dentro de ellos.

Este es un fragmento de Portada.xml. Es la definición de una ventana de portada que tiene, entre otras cosas, un botón de título Navegador con un nextLevel asociado a una ventana de contenedor web.

La etiqueta <nextLevelLevelId> tiene el identificador del level al que se irá si se pulsa el botón

La etiqueta <nextLevelDataId> tiene el identificador de los datos que el level debe cargar cuando se cree.

```
Portada.xml
<button>
        <buttonTitle>Navegador</buttonTitle>

<buttonFileName>images/buttonw.png</buttonFil
eName>
        <nextLevel>

<nextLevelLevelId>web</nextLevelLevelId>

<nextLevelDataId>web1</nextLevelDataId>
        </nextLevel>
</button>
```

Este es un fragmento del archivo app.xml donde se declaran los levels de nuestra aplicación.

Como solo tendrá ventanas de tipo web, solo aparece un level.

La etiqueta <levelId> contiene el identificador del level.

La etiqueta <levelFile> contiene el archivo xml que almacena los datos del level.

La etiqueta <levelType> contiene el tipo de ventana que corresponde al level (en este caso web).

```
App.xml
<levels>
        <!-- web -->
        <level>
        <levelId>web</levelId>
        <levelTitle>Ventana Web</levelTitle>
        <levelFile>web.xml</levelFile>
        <levelType>WEB_ACTIVITY</levelType>
        </level>
<levels>
```

Este es un fragmento del archivo de datos del level web. Por simplicidad se ha reducido el número de ventanas web de tres a dos.

La etiqueta <dataId> contiene el identificador de los datos dentro del archivo xml de datos.

La etiqueta <local> establece si hay que buscar una dirección en la web o en un archivo local.

La etiqueta <webUrl> contiene la dirección web o el archivo html local que cargará el contenedor web.

(Para más información de los tags de los archivos xml consulta Archivos xml).

```
Web.xml
<levelData>
        <data>
        <dataId>web1</dataId>
        <headerImageFile>images/web.png</headerImageFile>
        <headerText>Ventana Web</headerText>
        <local>true</local>
        <webUrl>archivo.html</webUrl>
        </data>
        <data>
        <dataId>web2</dataId>
        <headerImageFile>images/web.png</headerImageFile>
        <headerText>Ventana Web</headerText>
        <local>false</local>
        <webUrl>www.google.es</webUrl>

        </data>
</levelData>
```

Una vez explicados todos archivos xml que van a intervenir en el ejemplo, se mostrarán las interacciones entre ellos.

Actualmente estamos en el escenario en el que el usuario pulsa un botón de la portada que tiene asociado un nextLevel a una ventana web que cargará un archivo html local (archivo.html). En ese momento se busca el level en el app.xml por la etiqueta <nextLevelLevelId>.

```
Portada.xml
<button>
        <buttonTitle>Navegador</buttonTitle>

<buttonFileName>images/buttonw.png</buttonFileName>
        <nextLevel>
        <nextLevelLevelId>web</nextLevelLevelId>
        <nextLevelDataId>web1</nextLevelDataId>
        </nextLevel>
</button>
```

```
App.xml
<levels>
        <!-- web -->
        <level>
        <levelId>web</levelId>
        <levelTitle>Ventana
Web</levelTitle>
        <levelFile>web.xml</levelFile>

<levelType>WEB_ACTIVITY</levelType>
        </level>
<levels>
```

El level al que se ha hecho referencia dentro del app.xml, tiene asociado un archivo xml de datos en el tag <levelFile>. Se busca ese archivo y dentro de él aquellos datos a los que se hace referencia con el tag <nextLevelDataId> del archivo portada.xml.

```
Portada.xml
<button>
        <buttonTitle>Navegador</buttonTitle>

<buttonFileName>images/buttonw.png</buttonFileName>
        <nextLevel>
        <nextLevelLevelId>web</nextLevelLevelId>
                <nextLevelDataId>web1</nextLevelDataId>
        </nextLevel>
</button>
```

```
App.xml
<levels>
        <!-- web -->
        <level>
        <levelId>web</levelId>
        <levelTitle>Ventana
Web</levelTitle>
        <levelFile>web.xml</levelFile>

<levelType>WEB_ACTIVITY</levelType>
        </level>
<levels>
```

```
Web.xml
<levelData>
        <data>
        <dataId>web1</dataId>

<headerImageFile>images/web.png</headerImageFile>
        <headerText>Ventana Web</headerText>
        <local>true</local>
        <webUrl>archivo.html</webUrl>
        </data>
        <data>
        <dataId>web2</dataId>

<headerImageFile>images/web.png</headerImageFile>
        <headerText>Ventana Web</headerText>
        <local>false</local>
        <webUrl>www.google.es</webUrl>

        </data>
</levelData>
```

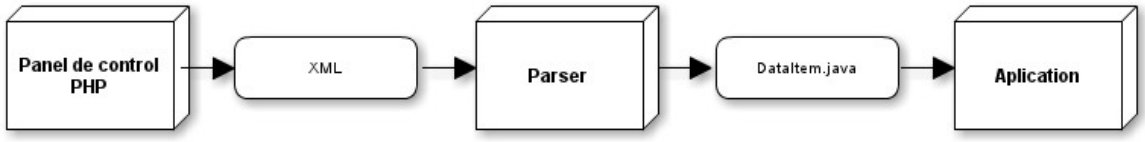
La segunda ventana cargará la dirección web www.google.es dentro del contenedor web. Hay que incluir en el archivo que define la portada (portada.xml) otro botón para que cargue los datos de la nueva ventana.

De esta manera, cada vez que se pulse el segundo botón, se accederá al mismo archivo de datos xml (web.xml) pero al nuevo identificar de datos que ahora contendrá la información para que se cargue google.

```
Portada.xml
<button>
        <buttonTitle>Navegador</buttonTitle>

<buttonFileName>images/buttonw.png</buttonFileName>
        <nextLevel>
        <nextLevelLevelId>web</nextLevelLevelId>
                <nextLevelDataId>web1</nextLevelDataId>
        </nextLevel>
</button>
<button>
        <buttonTitle>Navegador</buttonTitle>

<buttonFileName>images/buttonw.png</buttonFileName>
        <nextLevel>
        <nextLevelLevelId>web</nextLevelLevelId>
                <nextLevelDataId>web2</nextLevelDataId>
        </nextLevel>
</button>
```

AplicationData y AppLevelData.

Hasta ahora hemos visto que para consultar levels y sus archivos de datos xml, se accede al app.xml. Esto no es del todo cierto. Cuando se necesita acceder a los datos contenidos en un archivo xml, primero se parsea el archivo y se transforma en en un objeto que al que luego se puede acceder mediante código.



Como puede verse en la figura 2, los XML los genera el panel de control pero la aplicación en sí no accede a ellos. Necesita un objeto que almacene esa información y al que se pueda acceder desde código. Aunque realmente el parser es parte de la aplicación, trabaja como una entidad independiente y cuando es necesario genera estos objetos que la aplicación utilizará para construir las ventanas. (Ver parser).

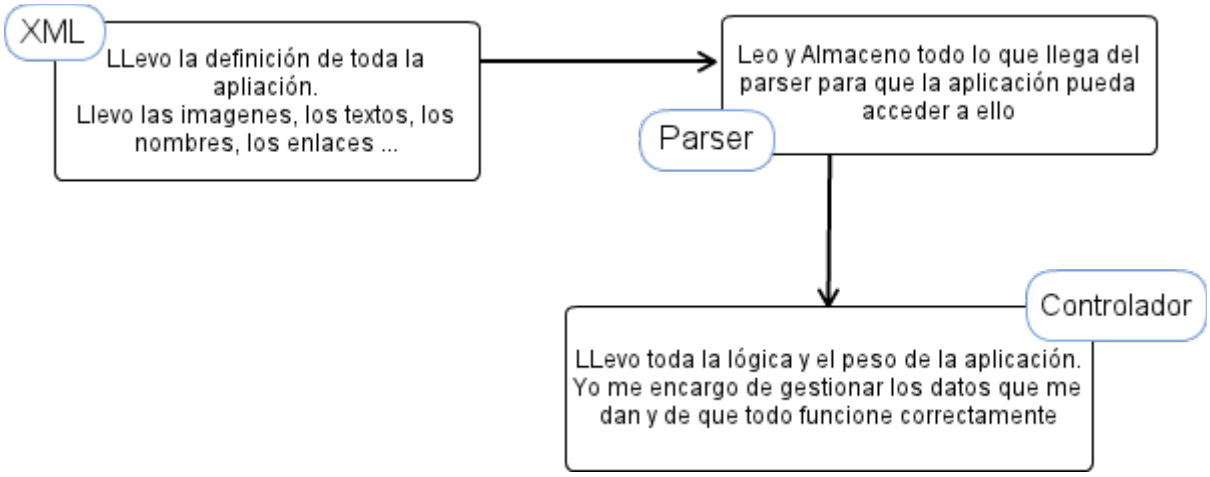
Aquí es donde entran en juego los **AplicationData** y los **AppLevelData**.

AplicationData : es el objeto que almacenará la información del app.xml. Previsiblemente habrá solo uno, aunque es posible generarlo manera dinámica en algunas ventanas como en los formularios.

AppLevelData : es el objeto que almacena la información de cada uno de los archivos xml. A su vez y como cada archivo de datos xml puede tener un conjunto de datos demasiado complejos para almacenarse en un tipo simple, es posible que sea necesario utilizar estructuras de datos complejas. Para estos nuevos tipos de datos se utilizan los DataItem.

Tipos de Actividades (pantallas)

Antes de pasar a explicar cada pantalla por separado así como las clases de las que se compone, vamos a explicar el flujo de ficheros que siguen cada una de ellas.



Bien ahora que ya sabemos que flujo siguen los distintos ficheros, estamos listos para pasar a ver cada pantalla por separado:

Pantalla de Splash

Pantalla de Splash	Esta pantalla aparece cuando arrancamos la aplicación, en lugar de mostrar directamente la portada, mostramos el splash para indicarle que la aplicación se está cargando y que tardará unos instantes

La pantalla de Splash no sólo nos puede servir para que la espera del usuario sea más amena, sino que también puede ser la presentación de nuestra aplicación. Para ello esta clase cuenta con una imagen que podremos mostrar para presentar nuestra aplicación.

El encargado de llamar a esta pantalla será el eMobcViewController, el jefe de nuestra aplicación, el controlador (ver la Importancia del eMobcViewController).

Pantalla de Cover

Pantalla de Cover	Esta es la pantalla de Portada. En la portada vamos a poder mostrar todas las opciones que queramos.

Dependiendo del tipo de aplicación que queramos hacer, nuestra portada puede ser de una manera o de otra.

Podemos decir que lo más importante de una portada son las opciones que esta lleva, tenemos que saber que opciones son las importantes y en qué orden se las podemos mostrar al usuario ya que eso puede darle diferentes matices a nuestra aplicación.

Como todo dentro del framework la portada también se genera dinámicamente, por lo que sólo tendremos que indicar las opciones que queremos que tenga y hacerlo en el orden adecuado que la portada ya se encargará de crear los botones que nos lleven a donde hemos indicado.

Para conseguir todo esto, para que la portada pueda manejar los datos que le hemos dado y que además los trate como hemos mencionado, necesitamos varios ficheros que vamos a nombrar y explicar a continuación :

Nombre Fichero	**Ubicación**	**Descripción**
cover.xml	XML Folder	Define los datos de la portada
coverParser	Parser Folder	Parsea cover.xml y almacena los datos definidos
Cover	Data Folder	Almacena todos los datos del XML
NwCoverController	UIClass Folder	Lleva toda la lógica y construcción de la pantalla

Ahora que ya conocemos las partes de las que se compone vamos a centrarnos un poco más en la explicación de cada una de ellas.

Cover.xml y CoverParser

Para saber más sobre la definición del xml o del funcionamiento del parser, ve a los capítulos de XML (cover.xml) y Parsers (CoverParser) respectivamente.

NwCoverController

Una vez llegados a NwCoverController es la hora de explicar la lógica de la pantalla de portada.

Si hemos llegado aquí sabemos que debemos tener los datos preparados para usar ya que sin ellos no podemos hacer nada. Estos datos (leídos y almacenados por el parser) se encuentran en una instancia de la clase Cover que hemos llamado *theCover.*

Tomando esos datos vamos a darle forma a la portada, es decir, vamos a crear los botones.

Para ello tenemos dos métodos uno para modo portrait y otro para modo landscape, pero ambos hacen lo mismo, crean los botones que anteriormente definimos formando dos columnas y posicionándolos de acuerdo a un determinado tamaño. No sólo los posiciona si no que los nombra y les pone la imagen que hemos elegido.

NwCoverController no sólo cuenta con los métodos propios del sistema y aquellos que crean botones, cuenta con otros métodos que se encargan de hacer efectiva la funcionalidad que le hemos dado a cada uno de los botones, es decir, van a ser los encargados de llevarme hacia las pantallas que corresponda en cada caso.

Pantalla de Galería de Imagenes

Pantalla de Galería de Imágenes	Esta pantalla muestra una galería de imágenes al usuario

Como bien indicamos anteriormente la galería de imágenes simplemente muestra al usuario las imágenes que hemos elegido. Para ello cuenta con la colaboraciçon de unas ficheros que ayudarán en su funcionamiento

Nombre Fichero	**Ubicación**	**Descripción**
image_gallery.xml	XML Folder	Define los datos de la galería de Imagenes
ImageGalleryParser	Parser Folder	Parsea image_gallery.xml y almacena los datos definidos
CachedImage	Cache Folder	Cachea las imagenes, las carga y las almacena
ImageGalleyItem	Form Folder	Define todos los atributos de un ImageGalleryItem
ImageGalleryLevelData	Form Folder	Contiene una array para almacenar los ImageGalleryItem
ImageGalleryLevel	Form Folder	Almacena todos los ImageGalleryItem en el array de la clase padre
NwImageGalleryController	UIClass Folder	Lleva toda la lógica y construcción de la pantalla

image_gallery.xml y ImageGalleryParser

Para saber más sobre la definición del xml o del funcionamiento del parser, ve a los capítulos de XML (image_gallery.xml) y Parsers (ImageGalleyParser) respectivamente.

Después de ver la tabla, podemos hacer una mención especial a la clase de CachedImage

CachedImage

Esta clase se encarga de tomar la ruta de la imagen (ya sea local o no) y de cargarla en el sistema.

Si es la primera vez que esa imagen aparece en el sistema, la cache no la contendrá por lo que tendremos que descargarla, en el caso de que no sea una imagen local, o simplemente acceder a ella, pero en ambos casos debemos almacenarla, asignarla a la instancia de UIImage que CachedImage contiene.

CachedImage no hace todo el trabajo por si sola, si no que utiliza otra clase que se encarga de llevar la lógica general. Para aclarar todo esto vamos a hacer un esquema y basarnos en el para que todo quede un poco más claro ya que CachedImage se utilizará siempre que queramos que nuestra aplicación cargue en una imagen.

Como podemos ver, para llevar el control de la cache del sistema contamos con 3 clases.

NwCache es la que contiene el mapa que se encarga de guardar los datos que ya han sido cacheadas. Ademas contiene metodos para acceder a datos almacenados, para guardar nuevos datos o para preguntar si ya se ha cacheado una ruta en concreto.

CacheContent contiene la ruta del fichero y además variables de control para indicar si el fichero es local o no o si ya se ha cacheado con anterioridad para así comportarse de una manera u otra. Para darle valor a estas variables de control, usa una instancia de la clase NwCache.

CachedImage sólo contiene un objeto image que será al que se le asigne la imagen cuando se carge o se obtenga de memoria. Para ello hereda de CachedContent y hace uso de todas las variables y métodos que le ofrece

Una vez explicado todo esto de manera general, puedes acceder a la documentación interna para saber más sobre la estructura de cache del sistema (Documentación Interna)

Pantalla de PDF

Pantalla de PDF	Esta pantalla muestra al usuario un lector de pdf

La pantalla de PDF es una de las más simples del sistema, simples en cuanto a su funcionamiento ya que lo único que hace es mostrar un lector de pdf al usuario.

Podemos destacar que los pdf se pueden encontrar bien alojados en el dispositivo o bien alojados en internet.

Pero a pesar de su sencillez cuenta con una serie de ficheros que ayudan a que por simple que sea su funcionamiento todo lo haga de la manera correcta. Estos son los siguientes:

Nombre Fichero	**Ubicación**	**Descripción**
pdf.xml	XML Folder	Define los datos del pdf
PdfParser	Parser Folder	Parsea pdf.xml y almacena los datos definidos
DataItem	Form Folder	Contiene los datos fijos de cada pantalla como por ejemplo el headerText
PdfLevelData	Form Folder	Contiene los datos que definen la pantalla de pdf
PdfLevel	Form Folder	Añada los dataItems al array de la clase padre (AppLevelData)
NwPdfController	UIClass Folder	Lleva toda la lógica y construcción de la pantalla

pdf.xml y PdfParser

Para saber más sobre la definición del xml o del funcionamiento del parser, ve a los capítulos de XML (pdf.xml) y Parsers (PdfParser) respectivamente.

Pantalla de lector QR

Pantalla QR	Esta pantalla permite al usuario escanear mediante la cámara un código QR

La pantalla QR es aquella que permite al usuario escanear códigos QR.

Esta pantalla no lleva mucha lógica por detrás, pero sí que necesita de unas librerías para poder funcionar.

Para ello hemos usado la librería de *ZBarSDK* que es la que va a tener realmente toda la lógica necesaria para poder escanear un código QR.

Pero a pesar de usar una librería de terceros que se encarga de la lógica de la pantalla, esta cuenta con una serie de ficheros que siguen la estructura del framework para poder funcionar que son los siguientes:

Nombre Fichero	**Ubicación**	**Descripción**
QR_lector.xml	XML Folder	Define los datos del QR
QRParser	Parser Folder	Parsea pdf.xml y almacena los datos definidos
QRLevelData	Form Folder	Contiene los datos que definen la pantalla de QR
QRLevel	Form Folder	-
NwQRController	UIClass Folder	Lleva toda la lógica y construcción de la pantalla

QR_lector.xml y QRParser

Para saber más sobre la definición del xml o del funcionamiento del parser, ve a los capítulos de XML (QR_lector.xml) y Parsers (QRParser) respectivamente.

Pantalla de Contenedor Web

Pantalla de Contenedor Web	Esta pantalla muestra al usuario un contenedor Web donde poder visualizar contenido on-line como páginas Web o HTML locales.

Como indica la descripción anterior, la pantalla del explorador Web permite al usuario visualizar una página web o un HTML local.

Para ello cuenta con un contenedor web que tanto podrá mostrar la URL que le indiquemos como podrá mostrar HTML que tengamos en local, incluso podemos hacer uso de estos últimos para incorporar css.

Veamos a continuación los ficheros necesarios para su funcionamiento:

Nombre Fichero	**Ubicación**	**Descripción**
web.xml	XML Folder	Define los datos del contenedor Web
WebParser	Parser Folder	Parsea web.xml y almacena los datos definidos
DataItem	Form Folder	Contiene los datos fijos de cada pantalla como por ejemplo el headerText
WebLevelData	Form Folder	Contiene los datos que definen el explorador Web
WebLevel	Form Folder	Añada los dataItems al array de la clase padre (AppLevelData)
NwWebController	UIClass Folder	Lleva toda la lógica y construcción de la pantalla

web.xml y WebParser

Para saber más sobre la definición del xml o del funcionamiento del parser, ve a los capítulos de XML (web.xml) y Parsers (WebParser) respectivamente.

Pantalla de Lista

Pantalla de Lista	Esta pantalla le muestra al usuario una lista donde. Permite seleccionar entre las opciones que muestra. Contamos con varios formatos de la misma pantalla: • Lista sólo con texto • Lista con imagen junto al texto

Como bien dice la pantalla anterior, la pantalla de lista muestra al usuario un conjunto de opciones en formato de lista.

Para mostrar la lista tenemos dos formatos:

- Por un lado, contamos con una lista que muestra sólo el texto, la descripción de las opciones
- Por otro lado, contamos con la misma lista con la descripción en cada celda pero además, en este caso, acompañamos cada descripción con una imagen.

Para dar forma a esta pantalla, se puede utilizar el mismo XML con la diferencia de que en el caso de que no tengamos imágenes dejemos este tag vacío, no lo indiquemos en el panel de control, pero si queremos tener imágenes el tag contendrá la ruta en local o la URL remota.

Hagamos aquí una pequeña pausa para explicar la carga de las imágenes en remoto en las celdas de la tabla

CARGA DE IMÁGENES REMOTAS

Para la carga remota de imágenes hemos optado por cargarlas de manera asíncrona para que de esta manera el usuario pueda usar la tabla sin tener que esperar a que se descarguen las imágenes de internet.

Para ello hemos utilizado una cola (NSOperationQueue) y una instancia de la clase (NSInvocationOperation) que define el método encargado de descargar las imágenes.

La carga la empezamos a hacer dentro del método ViewDidLoad que será el encargado de recorrer el array de items (procedente del data de la clase) y de encolar la operación en el caso de que la imagen a descargar no se encuentre previamente en el sistema.

Si la imagen no se encuentra en el sistema, encolamos la operación que empezará a descargar imágenes en paralelo mediante el método loadImage.

loadImage se encarga de descargar las imágenes, para ello hace uso de imageCached, que descarga la imagen desde la URL, además una vez descargada la guardará en el mapa del sistema para futuras referencias y además añadirá la imagen a una cola (imageToShow) de donde iremos tomando las imágenes que estén listas para ser mostradas.

Para mostrar las imágenes, es decir para poder asignarle la imagen a la celda, usamos el método displayImage.

displayImage se encarga de tomar la imagen de la cola, asignarla a la celda correspondiente y borrarla de la cola (imageToShow) para que de esta manera tome las imágenes en orden siguiendo el comportamiento de una cola.

Una vez explicado como funciona la carga de imágenes asíncronas, sólo cabe destacar que, al igual que la pantalla de Galeria de imagenes, también se utiliza la cache del sistema para controlar la carga de imagenes. Para saber más del funcionamiento de la cache ve al apartado de CachedImage.

Bien, veamos entonces que clases se van a encargar de la lógica y del traspaso de datos necesarios para el correcto funcionamiento de la lista

Nombre Fichero	**Ubicación**	**Descripción**
list.xml	XML Folder	Define los datos de la lista
ListParser	Parser Folder	Parsea list.xml y almacena los datos definidos
CachedImage	Cache Folder	Cachea las imágenes, las carga y las almacena
ListItem	Form Folder	Define todos los atributos de un ListItem
ListLevelData	Form Folder	Contiene una array para almacenar los ListItem
ListLevel	Form Folder	Almacena todos los ListLevelData en el array de la clase padre
NwListMultiLineCell	ListCells Folder	Da forma a una celda personalizada. Esta celda estará contenida por UITableViewCell
NwListController	UIClass Folder	Lleva toda la lógica y construcción de la pantalla

Cabe destacar que la pantalla de Listado también hace uso de la caché del sistema. Este punto ya ha quedado explicado en el apartado dedicado a la pantalla de galería de imágenes, si quieres saber como funciona la caché del sistema ve al apartado pantalla de Galería de Imágenes.

list.xml y ListParser

Para saber más sobre la definición del xml o del funcionamiento del parser, ve a los capítulos de XML (list.xml) y Parsers (ListParser) respectivamente.

Pantalla de Imagen + Lista

Pantalla de Imagen + Lista	Esta pantalla nos muestra una lista seleccionable por el usuario, acompañada por una imagen ajena a las celdas.

La pantalla muestra al usuario una imagen acompañada por una lista.

Esta pantalla sigue la misma lógica que la pantalla anterior a grandes rasgos. Las características de esta pantalla en concreto es que se va a mostrar una imagen en la parte superior de la pantalla y que las celdas de la lista sólo van a poder contener texto, no pueden ir acompañadas por otra imagen.

Para conseguir el funcionamiento de esta pantalla necesitamos los siguientes ficheros:

Nombre Fichero	Ubicación	Descripción
image_list.xml	XML Folder	Define los datos de la lista
ImageListParser	Parser Folder	Parsea image_list.xml y almacena los datos definidos
CachedImage	Cache Folder	Cachea las imágenes, las carga y las almacena
ListItem	Form Folder	Define todos los atributos de un ListItem
ImageListLevelData	Form Folder	Contiene una array para almacenar los ListItem y la imagen para mostrar
ImageListLevel	Form Folder	Almacena todos los ImageListLevelData en el array de la clase padre
NwListTextCell	ListCells Folder	Da forma a una celda personalizada. Esta celda estará contenida por UITableViewCell
NwImageListController	UIClass Folder	Lleva toda la lógica y construcción de la pantalla

Cabe destacar que la pantalla de imagen + texto también hace uso de la caché del sistema. Este punto ya ha quedado explicado en el apartado dedicado a la pantalla de galería de imágenes, si quieres saber como funciona la caché del sistema ve al apartado pantalla de Galería de Imágenes.

image_list.xml y ImageListParser

Para saber más sobre la definición del xml o del funcionamiento del parser, ve a los capítulos de XML (image_list.xml) y Parsers (ImageListParser) respectivamente.

Pantalla de Video

Pantalla de Video	Esta pantalla permite al usuario visualizar un video. El video puede estar en el propio dispositivo o puede ser on-line

Como se indica anteriormente la pantalla de video va a permitir al usuario acceder a videos alojados en local o remotos y poder reproducirlos en su dispositivo.

Debido a que podemos reproducir contenido local o remoto tenemos dos maneras diferentes de hacerlo, vamos a explicarlas por separado:

- **Video local**
  - Para reproducir contenido local, hemos utilizado una librería llamada MediaPlayer, que se encarga de tomar el contenido y reproducirlo

- **Video remoto**
  - Para reproducir contenido remoto, ya no podemos utilizar la librería anterior por lo que hemos creado nuestro propio método que se encarga de descargar y reproducir el contenido.

Una vez explicado esto vamos a ver que clases influyen en el funcionamiento de la pantalla de video:

Nombre Fichero	**Ubicación**	**Descripción**
video.xml	XML Folder	Define los datos de la pantalla de video
VideoParser	Parser Folder	Parsea video.xml y almacena los datos definidos
VideoLevelData	Form Folder	Contiene los datos que definen el reproductor de video
VideoLevel	Form Folder	Almacena todos los VideoLevelData en el array de la clase padre
NwVideoController	UIClass Folder	Lleva toda la lógica y construcción de la pantalla

video.xml y VideoParser

Para saber más sobre la definición del xml o del funcionamiento del parser, ve a los capítulos de XML (video.xml) y Parsers (VideoParser) respectivamente.

Pantalla de Mapa

Pantalla de Mapa	Esta pantalla muestra al usuario un mapa dependiendo de su localización. El mapa puede contener marcas de sitios preferidos que el usuario puede añadir

La pantalla de Mapa muestra al usuario un mapa de unas coordenadas específicas, por las que el usuario se puede mover para ver los alrededores a la posición dada. Además esta pantalla permite añadir marcas de sitios preferidos o lugares de interés que se cargarán a la vez que se carga el mapa.

El funcionamiento de esta pantalla es algo más complejo que el de las anteriores ya que tiene que cumplir con varios protocolos para su funcionamiento y además cuenta con la ayuda de una librería externa que se encarga del manejo de toda la lógica propia del mapa. Esta librería de terceros se llama BSForewardGeocorder.

A pesar de que la complejidad en cuanto a su funcionamiento sea un poco más elevada que las anteriores, cuenta con unas clases propias del framework que son equivalentes a todas las demás:

Nombre Fichero	**Ubicación**	**Descripción**
map.xml	XML Folder	Define los datos de la pantalla de video
MapParser	Parser Folder	Parsea map.xml y almacena los datos definidos
MapItem	Model Folder	Contiene todos los datos referentes las marcas que puede tener nuestro mapa.
MapLevelData	Form Folder	Contiene los datos que definen la pantalla de mapa.
MapLevel	Form Folder	Almacena todos los MapLevelData en el array de la clase padre
NwMapController	UIClass Folder	Lleva toda la lógica y construcción de la pantalla

map.xml y MapParser

Para saber más sobre la definición del xml o del funcionamiento del parser, ve a los capítulos de XML (map.xml) y Parsers (MapParser) respectivamente.

Pantalla de Búsqueda

Pantalla de Búsqueda	Esta pantalla permite al usuario buscar una porción de texto, georeferencia o imagen dentro del framework

Esta pantalla permite al usuario a través de una barra de búsqueda introducir una porción de texto, una georeferencia o imagen. La búsqueda se hará dentro de los archivos del framework, es decir dentro de nuestra propia aplicación.

Si se encuentra alguna coincidencia con el texto, georeferencia o imagen dada, se devolverán todas ellas para que el usuario seleccione la que buscaba.

Para conseguir que la pantalla funcione de manera correcta, necesita de los siguientes ficheros:

Nombre Fichero	**Ubicación**	**Descripción**
search.xml	XML Folder	Define los datos de la pantalla de búsqueda
SearchParser	Parser Folder	Parsea search.xml y almacena los datos definidos
SearchItem	Search Folder	Almacena un texto descriptivo y un NextLevel
SearchLevelData	Form Folder	*
SearchLevel	Form Folder	*
SearchListCell	ListCells Folder	Da forma a una celda personalizada. Esta celda estará contenida por UITableViewCell
NwSearchController	UIClass Folder	Lleva toda la lógica y construcción de la pantalla

search.xml y SearchParser

Para saber más sobre la definición del xml o del funcionamiento del parser, ve a los capítulos de XML (search.xml) y Parsers (SearchParser) respectivamente.

Pantalla de Formulario

Pantalla formulario, Se encarga de recoger datos y guardarlos o enviarlos. Pudiendo utilizarlos en la aplicación si hace falta.No sirve para crear nuevas pantallas a partir de ella.

Pantalla Formulario	Esta pantalla permite al usuario crear una nueva pantalla simplemente rellenando los campos necesarios

Esta pantalla es la pantalla más especial y potente del framework ya que permite al usuario desde una pantalla crear otra, es decir, dada un pantalla con unos campos predefinidos, bastará con que rellene los campos para que se cree una nueva pantalla acorde a los datos introducidos.

Para conseguir la lógica adecuada, se necesitan los siguientes archivos:

Nombre Fichero	**Ubicación**	**Descripción**
form.xml	XML Folder	Define los datos de la pantalla de formulario
FormParser	Parser Folder	Parsea form.xml y almacena los datos definidos
FormLevelData	Form Folder	Contiene los datos que definen la pantalla de formulario.
FormLevel	Form Folder	Almacena todos los FormLevelData en el array de la clase padre
NwFormController	UIClass Folder	Lleva toda la lógica y construcción de la pantalla

Podemos decir que ya que estamos ante una pantalla cuyo funcionamiento difiere tanto del resto de pantallas del framework, tenemos los mismos ficheros para su funcionamiento.

Realmente no son necesarios nada más que dichos ficheros ya que todo sigue la misma estructura, pero podemos decir que dentro de eMobcViewController así como dentro de NwUtils hay métodos especiales para esta pantalla que se encargan de cargar el nextLevel creado desde el formulario.

Si quieres saber más hacemos una mención especial de este tema en los capítulos dedicados exclusivamente para explicar los ficheros de eMobcViewController y NwUtils.

form.xml y FormParser

Para saber más sobre la definición del xml o del funcionamiento del parser, ve a los capítulos de XML (form.xml) y Parsers (FormParser) respectivamente.

Pantalla de Imagen + Texto

Pantalla Imagen + Texto	Esta pantalla muestra al usuario una Imagen acompañado por un texto

Esta pantalla es una pantalla explicativa o informativa para el usuario. En ella va a aparecer una imagen en la parte superior o inferior de la pantalla (donde esté definida desde el panel de Control) acompañada por un texto que puede o no tener relación con la imagen mostrada.

Para el funcionamiento de esta pantalla usamos los siguientes ficheros:

Nombre Fichero	**Ubicación**	**Descripción**
image_description.xml	XML Folder	Define los datos de la pantalla de imagen + texto
ImageDescriptionParser	Parser Folder	Parsea image_description.xml y almacena los datos definidos
CachedImage	Cache Folder	Cachea las imagenes, las carga y las almacena
ImageTextDescriptionLevelData	Form Folder	Contiene los datos que definen la pantalla de imagen + texto.
ImageTextDescriptionLevel	Form Folder	Almacena todos los ImageTextDescriptionLevelData en el array de la clase padre
NwImageTextController	UIClass Folder	Lleva toda la lógica y construcción de la pantalla

Cabe destacar que la pantalla de imagen + texto también hace uso de la caché del sistema. Este punto ya ha quedado explicado en el apartado dedicado a la pantalla de galería de imágenes, si quieres saber como funciona la caché del sistema ve al apartado pantalla de Galería de Imágenes.

image_description.xml y ImageDescriptionParser

Para saber más sobre la definición del xml o del funcionamiento del parser, ve a los capítulos de XML (image_description.xml) y Parsers (ImageDescriptionParser) respectivamente.

Pantalla de Imagen con Zoom

Pantalla de Imagen con Zoom	Esta pantalla permite mostrar al usuario una imagen aumentada

Esta pantalla permite que el usuario pueda ver una imagen con gran Zoom, con una gran resolución. Debido a que la imagen es mayor que una imagen que se pueda mostrar en la galería, contamos con un contenedor 'especial' que permita mostrar la imagen en tamaño real y que además le permite al usuario 'navegar' a través de la imagen para poder ver todas sus partes a un tamaño real.

En el funcionamiento de esta pantalla influyen los siguientes archivos:

Nombre Fichero	**Ubicación**	**Descripción**
image_zoom.xml	XML Folder	Define los datos de la pantalla de imagen con Zoom
ImageZoomParser	Parser Folder	Parsea zoom_image.xml y almacena los datos definidos
CachedImage	Cache Folder	Cachea las imágenes, las carga y las almacena
ImageZoomLevelData	Form Folder	Contiene los datos que definen la pantalla de imagen con zoom
ImageZoomLevel	Form Folder	Almacena todos los ImageZoomLevelData en el array de la clase padre
NwImageZoomController	UIClass Folder	Lleva toda la lógica y construcción de la pantalla

Cabe destacar que la pantalla de imagen con Zoom también hace uso de la caché del sistema. Este punto ya ha quedado explicado en el apartado dedicado a la pantalla de galería de imágenes, si quieres saber como funciona la caché del sistema ve al apartado pantalla de Galería de Imágenes.

image_zoom.xml y ImageZoomParser

Para saber más sobre la definición del xml o del funcionamiento del parser, ve a los capítulos de XML (image_zoom.xml) y Parsers (ImageZoomParser) respectivamente.

Pantalla de Multimedia

Pantalla de Multimedia	Esta pantalla permite mostrar al usuario un menú para acceder a las acciones multimedia como pueden ser: • Acceso a Fotos • Acción de Compartir • Acceso a Videos • Acceso a grabaciones de Voz • ...etc...

Esta pantalla permite al usuario acceder a un menú multimedia desde donde poder acceder a opciones multimedia.

Como opciones multimedia podemos entender acciones como compartir, acceder a fotos, a videos, a grabaciones de voz ...

Para lograr el funcionamiento de esta pantalla son necesarios los siguientes ficheros:

Nombre Fichero	**Ubicación**	**Descripción**
multimedia.xml	XML Folder	Define los datos de la pantalla de multimedia
MultimediaParser	Parser Folder	Parsea multimedia.xml y almacena los datos definidos
MultimediaLevelData	Form Folder	Datos de Multimedia
MultimediaLevel	Form Folder	Nivel de Multimedia
NwMultimediaController	UIClass Folder	Lleva toda la lógica y construcción de la pantalla

multimedia.xml y MultimediaParser

Para saber más sobre la definición del xml o del funcionamiento del parser, ve a los capítulos de XML (multimedia.xml) y Parsers (MultimediaParser) respectivamente.

Pantalla de Calendario

Pantalla de Calendario	Esta pantalla muestra al usuario un calendario ajeno al calendario del sistema. Este calendario muestra los eventos que la aplicación internamente quiera mostrar.

Esta pantalla permite mostrar al usuario un calendario ajeno al calendario del sistema, es decir, es un calendario que solamente va a mostrar eventos propios de la aplicación, lo que el desarrollador haya fijado desde el panel de control.

Este calendario no le va a permitir añadir eventos al usuario ya que no tiene relación con el del sistema, sólo va a poder interactuar con los eventos que se hayan fijado desde el panel de control. Se va a poder mover por los meses del calendario y acceder a aquellos días que muestren eventos así como poder acceder a una información más detallada de ellos.

La pantalla del calendario es otra pantalla especial del framework ya que necesita de una librería externa que va a hacer el trabajo de dibujar y de llevar la lógica del calendario.

Esta librería se llama Kal. Kal contiene clases que se encargan de llevar la cuenta de los días, de navegar entre los distintos meses y además contiene la lógica y las imágenes necesarias para dibujar el calendario.

Pero esta pantalla no es especial sólo por usar una librería que haga todo eso por nosotros si no porque a diferencia de las demás esta usa más de un ViewController para controlar la vista, es decir, no sólo vamos a usar el controller propio del framework si no que hemos asociado un objeto de tipo UIWindow al ViewController que nos proporciona la librería.

A pesar de todo ello, estos son los ficheros relacionados con la ventana de calendario:

Nombre Fichero	**Ubicación**	**Descripción**
calendar.xml	XML Folder	Define los datos de la pantalla de calendario
CalendarParser	Parser Folder	Parsea calendar.xml y almacena los datos definidos
AppEvent	Model Folder	Contiene todos los datos relacionados con un evento
CalendarLevelData	Form Folder	Almacena todos los eventos e implementa la lógica del calendario
CalendarLevel	Form Folder	Almacena todos los CalendarLevelData en el array de la clase padre
NwCalendarController	UIClass Folder	Lleva toda la lógica y construcción de la pantalla
KalViewController	Lib/Kal Folder	Lleva toda la lógica interna calendario. Además se encarga de dibujarlo de manera correcta

calendar.xml y CalendarParser

Para saber más sobre la definición del xml o del funcionamiento del parser, ve a los capítulos de XML (calendar.xml) y Parsers (CalendarParser) respectivamente

CalendarLevelData

Si habeís echado un vistazo al código de la pantalla de calendario, habreis visto que el LevelData de esta pantalla es completamente diferente que el LevelData de las demás, por eso hemos creído importante explicar con un poco mas de detalle esta clase.

Como ya hemos comentado anteriormente, la pantalla de calendario necesita una librería llamada Kal, esta librería es la que contiene toda la lógica que necesita el calendario, si es la librería la que se encarga de manejar el calendario, ¿por qué decimos que CalendarLevelData va a llevar la lógica?

Expliquémoslo, la librería contiene unos métodos que los llama desde abajo (callbacks), estos métodos están sin

implementar ya que la implementación varía dependiendo de las estructura de datos que se usen, debido a ello hemos tenido que implementar esos métodos para que Kal pudiera tomar los datos de la manera que necesita acorde a como los tenemos nosotros almacenados.

Para almacenar los datos contamos con las siguientes estructuras:

AllEvents	Diccionario que almacena todos los eventos presentes en calendar.xml Este diccionario almacena los eventos de la siguiente manera: • clave, toma la fecha (entera) del evento (17/07/2012) • valor, lista con todos los eventos que tenga la fecha de la clave Este diccionario se usará como base de datos del calendario donde van a estar presentes todos los eventos
MonthEvents	Diccionario que almacena todos los eventos que estén dentro de un determinado mes. Este diccionario almacena los eventos de la siguiente manera: • clave, toma el día del evento (17) • valor, lista con todos los eventos que tenga el día de la clave Este diccionario se usa para tomar los eventos del mes que se está mostrando en la pantalla y poder mostrarlos
DayEvents	Lista de eventos de un determinado día. Esta lista se usa para mostrar en la lista los eventos que estén dentro del día seleccionado

Una vez vistas las diferentes estructuras de datos que hemos usado vamos a ver los métodos mas representativos y vamos a explicarlos de manera general:

LoadItemsFromDate:fromDate:toDate	Este método va a cargar dentro de dayEvents la lista de eventos que se corresponda con el valor de la clave dada. La clave es el día que le entra por fromDate
LoadEventsFrom:fromDate:toDate	Este método va a consultar nuestra base de datos allEvents y va a guardar dentro de monthEvents todos aquellos eventos que esten dentro del intervalo de fechas dado con fromDate y toDate

Además contamos con métodos que vuelven a inicializar monthEvents y dayEvents que solo se llaman cuando cambiamos de mes dentro de la vista. Esto se hace para que los eventos que se muestran se correspondan al mes que esta seleccionado. Además contiene métodos del protocolo de UITableView que se encargan de manejar la lista donde se muestran los eventos.

Cómo Crear una Nueva Pantalla en IOS

Antes de empezar con el manual de como crear una nueva pantalla en IOS, vamos a puntualizar que este manual no incluye la lógica de la pantalla sólo incluye los pasos para conseguir que nuestro framework tenga y pueda cargar una nueva pantalla.

Para crear un nuevo estilo de pantalla lo primero que tenemos que hacer es una pantalla que esté vacía, es decir, que sólo tenga lo básico para poder funcionar.

Para nuestro ejemplo vamos a tomar una plantilla que se puede corresponder con una de las pantallas básicas.

Pasos para crear una nueva pantalla de IOS en el framework:

Crear los archivos correspondientes para guardar los datos de nuestra aplicación.

Puesto que contienen los datos de la aplicación, es dentro de este fichero donde tenemos que declarar todos los datos que nuestra aplicación va a necesitar para funcionar. Estos archivos deben estar dentro de la carpeta DATA y tendrán los siguientes nombres:

*nuevaPantallaLevelData.h

*nuevaPantallaLevelData.m

```
#import <Foundation/Foundation.h>
#import "DataItem.h"

@interface NuevaPantallaLevelData : DataItem {
    NSString* npData1;
    NSString* npData2;
    NSString* npData3;
}

@property (nonatomic, copy) NSString* npData1;
@property (nonatomic, copy) NSString* npData2;
@property (nonatomic, copy) NSString* npData3;


@end
```

Crear los nuevos niveles para nuestra pantalla.

El Level será el encargado de añadir los datos de la pantalla en el array de la clase padre, de la clase que contiene los datos de la aplicación.

En nuestro caso sólo contendrá un método (como todos los Level) que los añadirá a una lista y a un diccionario de la aplicación

Nuestros Archivos están en la carpeta LEVEL con los nombres de:

nuevaPantallaLevel.h

nuevaPantallaLevel.m

```
#import "NuevaPantallaLevel.h"

- (void) addLevelDataItem:(NuevaPantallaLevelData*) newDataItem{
    [super addItem:(DataItem*)newDataItem];;
}
```

Crear el xib de nuestra pantalla.

Es decir, tenemos que crear el aspecto de nuestra pantalla. Siguiendo el razonamiento de ir copiando los archivos que dan forma a una plantilla básica bastará con copiar el mismo xib y pegarlo dentro de la carpeta XIB.

A la hora de personalizar nuestra pantalla tendremos que incluir en el xib aquellos componentes que necesitemos para crear el aspecto de la pantalla que necesitemos.

El nuevo archivo tendrá el nombre de:

NwNuevaPantallaController.xib

Crear el nuevo controlador para que dirija nuestra interfaz.

Debido a que hemos creado un nuevo xib, le tenemos que asociar a un nuevo controlador.

El controlador es quién se encarga de llevar toda la lógica de la pantalla.

Como queremos que crear una nueva pantalla tenemos que crear un nuevo controlador aunque de momento sólo vamos a copiar el controlador de la pantalla de plantilla renombrandolo para que nos quede de la siguiente manera:

NwNuevaPantallaController.h

NwNuevaPantallaController.m

Si queremos personalizar nuestra pantalla tendremos que incluir dentro de estos archivos los métodos necesarios para que la nueva pantalla se comporte como queremos.

En este punto no sólo vamos a tener que crear los ficheros de los controladores si no que le tenemos que decir al sistema que cuando cargue la Vista de nuestra pantalla cargue también su controlador.

Para ello tenemos que abrir el xib y abrir el Inspector sobre el File's Owner.

Cuando lo tengamos abierto nos vamos a la última pestaña, Identity (**comando-4**) y cambiamos la pestaña de clase donde le vamos a poner el nuevo controlador que hemos creado. De esta manera lo que estamos haciendo es decirle al Owner de nuestra Vista que necesita de nuestro nuevo controlador para funcionar.

El File's Owner es el encargado de cargar el xib. Podemos decir que es el que enlaza la parte gráfica de nuestra pantalla con la parte de código bien sea el controlador como los eventos (IBAction) y los componentes (IBOutLet)

Modificar el archivo de AppLevel.h

AppLevel contiene todos los tipos de pantalla que hasta ahora soportaba el framework, para que el framework reconozca el nuevo tipo de pantalla que estamos creando tenemos que añadir a la variable de tipo enumerado ActivityType el nuevo tipo que representará a la nueva pantalla en nuestro caso sería así:

NUEVAPANTALLA_ACTIVITY

```
typedef enum ActivityType {
    COVER_ACTIVITY,
    IMAGE_TEXT_DESCRIPTION_ACTIVITY,
    IMAGE_LIST_ACTIVITY,
    LIST_ACTIVITY,
    VIDEO_ACTIVITY,
    IMAGE_ZOOM_ACTIVITY,
    IMAGE_GALLERY_ACTIVITY,
    BUTTONS_ACTIVITY,
    FORM_ACTIVITY,
    MAP_ACTIVITY,
    WEB_ACTIVITY,
    PDF_ACTIVITY,
    QR_ACTIVITY,
    SPLASH_ACTIVITY,
    SEARCH_ACTIVITY,
    CALENDAR_ACTIVITY,
    NUEVAPANTALLA_ACTIVITY
}ActivityType;
```

De esta manera cuando queramos navegar hasta nuestra nueva pantalla podremos reconocerla e invocar a los métodos específicos que me indicaran el controlador a usar, el parser o el xml asociados

Ahora nos vamos a la clase nwUtil

Esta clase es la que se encarga de dado el tipo de actividad invocar al método que indica el parser a utilizar, para conseguir esta funcionalidad tenemos que seguir los siguientes pasos:

- Lo primero que tenemos que hacer es irnos a nwUtil.h y declarar un nuevo método que nos ayudará a elegir el parser adecuado para nuestra pantalla, el propio de la pantalla.
    - readNuevaPantallaData
- Después de haber añadido el nuevo método en la cabecera de nuestro archivo, vamos a seguir estos dos pasos:
    - Implementar el método readNuevaPantallaData, que será el encargado de cargar todos los datos del xml en la nueva pantalla (para ello para empezar y puesto que todos siguen la misma estructura, podemos copiar el método correspondiente a nuestra plantilla)

```
-(NuevaPantallaLevelData*) readNuevaPantallaData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel {
    NuevaPantallaParser *parser = [[NuevaPantallaParser alloc] init];

    NSData* parseData = [appLevel.file content];
    [parser parseXMLFileFromData:parseData];

    NuevaPantallaLevel* theLevel = (NuevaPantallaLevel*)parser.parsedLevel;

    DataItem* theItem = nil;
    if (nextLevel.dataId != nil) {
        theItem = [theLevel dataItemById:nextLevel.dataId];
    }else {
        theItem = [theLevel dataItemByNumber:nextLevel.dataNumber];
    }

    [parser release];
    return (NuevaPantallaLevelData*)theItem;
}
```

- Añadir al método readAppLevelData la opción necesaria para cuando le entre nuestro tipo de pantalla (NUEVAPANTALLA_ACTIVITY) pueda llamar al método antes implementado.

```
-(DataItem*) readAppLevelData:(NextLevel*)nextLevel {

    [. . .]

    switch (theLevel.type) {

        [. . .]

        case QR_ACTIVITY:
            theData = [self readQRData:theLevel nextLevel:nextLevel];
            break;
        case BUTTONS_ACTIVITY:
            theData = [self readButtonsData:theLevel nextLevel:nextLevel];
            break;
        case FORM_ACTIVITY:
            theData = [self readFormData:theLevel nextLevel:nextLevel];
            break;
        case LIST_ACTIVITY:
            theData = [self readListData:theLevel nextLevel:nextLevel];
            break;
        case CALENDAR_ACTIVITY:
            theData = [self readCalendarData:theLevel nextLevel:nextLevel];
            break;
        case NUEVAPANTALLA_ACTIVITY:
            theData = [self readNuevaPantallaData:theLevel nextLevel:nextLevel];
            break;
        default:
            break;
    }
    [util release];
    [theLevel release];
    return theData;
}
```

El eMobcViewController es el encargado de invocar al método adecuado para poder cambiar de nivel, es decir cambiar de pantalla y asociar el controlador adecuado.

Para conseguir que el eMobcViewController pueda llamar al controlador de nuestra nueva pantalla tenemos que crear los siguientes métodos con su correspondiente definición en la cabecera.

Puesto que la estructura de estos métodos es la misma prácticamente para todos podemos tomar los métodos

correspondientes a nuestra plantilla y ajustarlos a la nueva pantalla

- Dentro de eMobcViewController tenemos que definir e implementar los siguientes métodos. Para ello y puesto que la estructura es semejante en todos podemos tomar los métodos asociados a la plantilla y cambiar aquello que sea específico a de cada pantalla.
    - loadNuevaPantallaNextLevel, carga el nuevo nextLevel y su correspondiente controlador, es decir, invoca a loadNuevaPantallaController

```
- (void) loadNuevaPantallaNextLevel:(LoadUtil*) util{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    NuevaPantallaLevelData* theData = [[NwUtil instance] readNuevaPantallaData:util.appLevel
                                                                     nextLevel:util.nextLevel];

    LoadControlerUtil* controlerUtil = nil;
    controlerUtil = [[LoadControlerUtil alloc] initWithValues:util.appLevel
                                                       theData:theData];

    [self performSelectorOnMainThread:@selector(loadNuevaPantallaController:)
                           withObject:controlerUtil
                        waitUntilDone:NO];

    [controlerUtil release];
    [pool drain];
}
```

    - loadNuevaPantallaController, carga el nuevo controlador

```
- (void) loadNuevaPantallaController:(LoadControlerUtil*)  theControlerUtil{
    NSString* controllerNibName = [eMobcViewController addIPadSuffixWhenOnIPad:@"NwNuevaPantallaController"];
    NwNuevaPantallaController *controller = [[NwNuevaPantallaController alloc] initWithNibName:controllerNibName
                                                                                        bundle:nil];
    controller.data = (NuevaPantallaLevelData*)theControlerUtil.data;
    controller.mainController = self;

    self.currentController = controller;

    [self loadController:currentController withAppLevel:theControlerUtil.appLevel];
    [controller release];

}
```

- Para conseguir que se llame al NextLevel de nuestra pantalla y que se le asocie el controlador adecuado tenemos que modificar:
    - showNextLevel, dentro de este métodos tenemos que añadir otra opción dentro del case que soporte el nuevo tipo de actividad y que cuando le entre este tipo invoque al método de loadNuevaPantallaNextLevel

```
- (bool) showNextLevel:(NextLevel*) nextLevel{

        [. . .]

        switch (appLevel.type) {

            [. . .]

            case PDF_ACTIVITY:
                loadSelector = @selector(loadPdfNextLevel:);
                break;
            case QR_ACTIVITY:
                loadSelector = @selector(loadQRNextLevel:);
                break;
            case CALENDAR_ACTIVITY:
                loadSelector = @selector(loadCalendarNextLevel:);
                break;
            default:
                break;
        }

        [. . .]
}
```

Llegados a este punto tenemos que modificar el app.xml añadiendo la nueva actividad, es decir, sabemos que el app.xml es el xml que va a tener la información de todo (o casi todo) lo que puede hacer el framework.

No sólo eso sino que es al app.xml donde vamos a ir a buscar los demás ficheros xml cuando queramos cargar un nuevo nextLevel.

Debido a ello tenemos que añadir un nuevo nivel donde tenemos que indicar el nuevo tipo de actividad (NUEVAPANTALLA_ACTIVITY),el levelID, y el levelFile que será el nombre del archivo xml asociado con nuestra nueva pantalla.

```
<level>
    <levelId>Nueva Pantalla</levelId>
    <levelTitle>Nueva Pantalla</levelTitle>
    <levelFile>nueva_pantalla.xml</levelFile>
    <levelType>NUEVAPANTALLA_ACTIVITY</levelType>
    <levelXib/>
    <levelTransition>None</levelTransition>
</level>
```

Modificado el app.xml tenemos que tomar el nombre que le hemos puesto dentro de la etiqueta levelFile y crear así el xml que va a definir los datos de nuestra pantalla.

Como lo que estamos haciendo simplemente es incluir una nueva pantalla en el framework y siguiendo el procedimiento de tomar la plantilla para conseguir una rápida integración, en este caso sólo será necesario tomar el xml asociado a nuestra plantilla. Podemos llamar al nuevo xml de la siguiente manera:

- nueva_pantalla.xml

Tenemos que tener en cuenta que cuando queramos personalizar nuestra pantalla tendremos que crear un xml ajustado a sus necesidades, de tal manera que podamos definir una nueva pantalla con los datos correspondientes.

Cabe destacar que los xml no son implementados por nosotros a mano, si no que son generados por el panel de control.

Llegados a este punto y recien modificado el app y creado nueva_pantalla.xml, tenemos que modificar AppParser (asociado al zpp.xml) y además tenemos que crear un nuevo parser para que pueda leer el xml que tendrá nuestra nueva pantalla.

- AppParser, será el encargado de parsear app.xml por lo que si hemos añadido nuevas etiquetas (en este caso el nuevo tipo de actividad) se lo tendremos que indicar, para que cuando lo lea pueda reconocerla, así que le tendremos que añadir esta nueva constante como:
    - kActTypeNuevaPantalla = @"NUEVAPANTALLA_ACTIVITY"

```
[. . .]

static NSString * const kActTypeWeb = @"WEB_ACTIVITY";
static NSString * const kActTypePdf = @"PDF_ACTIVITY";
static NSString * const kActTypeQR = @"QR_ACTIVITY";
static NSString * const kActTypeCalendar = @"CALENDAR_ACTIVITY";
static NSString * const kActTypeNuevaPantalla = @"NUEVAPANTALLA_ACTIVITY";

[. . .]
```

NuevaPantallaParser, este será el parser que esté asociado a la nueva pantalla. Debido a ello este parser tiene que contener una constante por cada etiqueta del xml que queramos reconocer, y una acción que se corresponda con la lectura del valor de la etiqueta.

Como lo que estamos haciendo es una rápida integración, bastará con copiar el parser asociado a nuestra plantilla. No debemos olvidar que si queremos personalizar la pantalla vamos a tener que crear un nuevo parser que sea capaz de leer el xml específico de la nueva pantalla. Para crearlo, tenemos que saber que todos los parser siguen una estructura fija por lo que conviene seguirla, para saber más acerca de las estructuras de los parser, ve al capítulo de Estructura de Parser.

Si hemos seguido todos los pasos anteriores y si no hemos tocado el código de todos los ficheros que hemos copiado tendremos que tener una copia de la pantalla que hemos tomado como plantilla sólo que con diferente nombre.

Para ver si realmente nuestra pantalla ha quedado integrada (aunque de momento sea una copia de otra) sólo tendremos que modificar (por ejemplo) el xml de la portada para que sea capaz de cargar como nextLevel nuestra nueva pantalla.

De ser así:

Felicidades, acabas de integrar tu primera pantalla en IOS

Aunque hayas creado una nueva pantalla, siguiendo estos pasos te has asegurado de que el framework la reconoce y que vas por buen camino.

Ahora sólo tienes que cambiar los ficheros que has copiado e implementarlos por tí mismo para que la pantalla haga lo que tu quieras que haga, pero no se te olvide tener en cuenta que

(por norma general):

- el archivo Level, sólo va a contener un método que añadirá el nivel propio a la pila del sistema.
- el archivo Data va a contener los datos de la aplicación, es decir, todas las etiquetas del xml que puedan servir o que sean necesarios para almacenar datos tendrán que ser declaradas dentro de estos ficheros, ya que el parser va a necesitar algún “sitio” para almacenar los datos que lea, así como la lógica que permite manejarlas. Es decir, si queremos que nuestra pantalla guarde los datos en una lista, tendremos que declararla en Data y además especificar el método de añadir elementos a esa lista.
- el archivo Controller, va a contener toda la lógica que va por debajo de la parte GRÁFICA de la pantalla. El puede manejar los datos que el parser almacenó en el fichero DATA y además será el encargado de hacer que esos datos se muestren en los componentes o de que estos puedan interactuar con ellos.
- Por último hay que tener especial cuidado a la hora de diseñar el xml y su correspondiente parser, ya que estos tienen que ir “enlazados” y todo lo que se ponga en el xml tiene que ser reconocido por el parser.

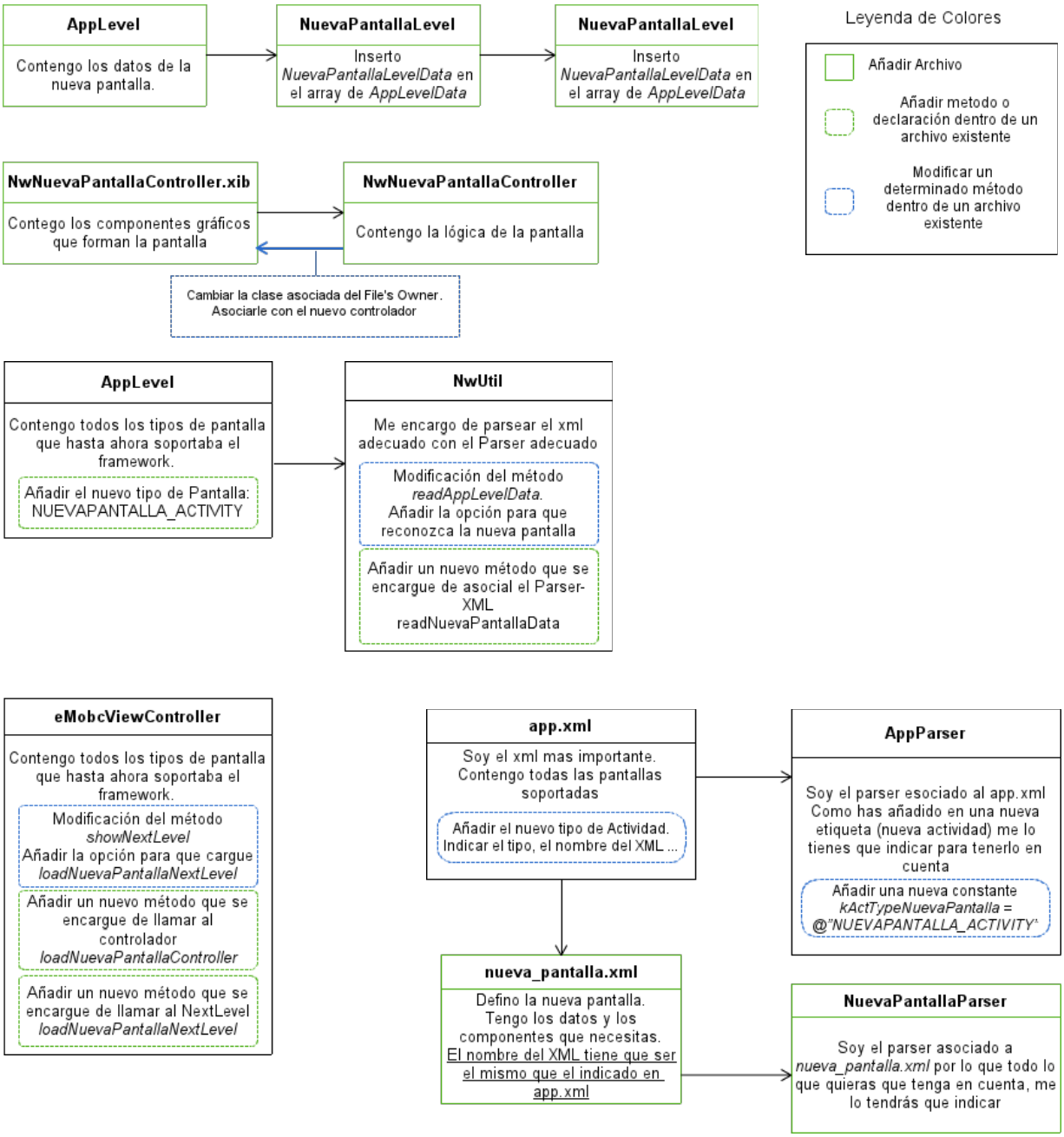
Cómo Crear una Nueva Pantalla (esquemático) en IOS

Para crear una pantalla en IOS hay que seguir unos pasos fijos que los vamos a ilustrar de manera esquemática, si quieres saber un poco más o ir más despacio ve a

Como Crear Una Nueva Pantalla en IOS (Extendido)

Antes de empezar vamos a decir que al igual que en la versión extendida, para una rápida integración en el framework, vamos a utilizar una pantalla de plantilla de la que vamos a copiar todas sus partes, modificando las partes especiales para cada pantalla, para que de esta manera la integración sea más rápida y efectiva.

Empecemos:



Descripción de los Parser IOS

Los parser son los ficheros que se encargan de leer los ficheros xml.

Su meta es tomar el contenido de cada una de las etiquetas y dependiendo de la etiqueta que sea guardarlo donde corresponda. Para ello tenemos un parser por cada xml, ya que cada uno va a tener que reconocer etiquetas específicas, que van a tener un significado concreto dependiendo del tipo de pantalla que definan.

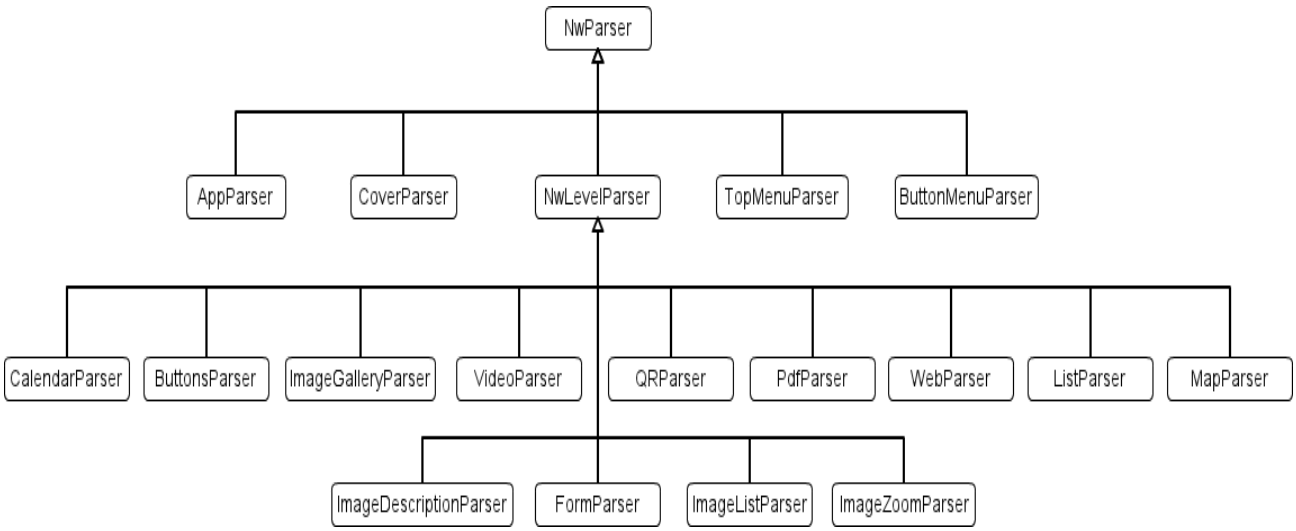
Bien ahora que ya sabemos como son los parser, vamos a explicar cada uno de ellos.

Para ello primero vamos a explicar de manera general **NwParser**.

NwParser

Podemos decir que NwParser no es un parser como los demás que podamos encontrarnos en el framework ya que para empezar no parsea ningún XML, sino que por el contrario, va a definir variables y métodos base que serán la base de nuestro parser. Estos métodos y atributos los podrán utilizar aquellas clases que hereden de NwParser.

Estas son las relaciones de herencia que existen entre los ficheros de parser:



Como podemos ver NwParser es el padre (y el abuelo) de todos los demás parser del framework. El más importante es NwLevelParser que es el parser del que heredan todas los parser que definen una pantalla.

También se puede observar que hay 4 parser más, AppParser, CoverParser, TopMenuParser y ButtonMenuParser que heredan directamente de NwParser, estos cuatro van a ser parser especiales cuyo funcionamiento ya explicaremos más adelante.

Aunque cada parser sirva para parsear una pantalla, un tipo de xml, todos van a seguir la misma estructura. Debido a ello antes de meternos con la explicación general de cada parser, vamos a centrarnos en la estructura que todo los parser siguen

- EjemploParser.h

En el archivo de cabecera de cada parser vamos a encontrarnos los LevelData a los que este esté asociado.

Además vamos a encontrar una instancia de NextLevel siempre que el parser se corresponda con una pantalla que permite al usuario navegar hacia dentro de la aplicación como por ejemplo una botonera, pantallas como el lector de pdf carecen de navegación hacia dentro por lo que no habrá declaración de NextLevel en su parser

- EmjemploParser.m

Dentro de este archivo va a estar toda la lógica del parser, esta es bastante fácil y la podemos dividir en tres secciones:

Declaración de constantes -> etiquetas del xml

Dentro de este apartado tienen que estar declaradas todas las etiquetas que tenga nuestro xml, que son las etiquetas que va a reconocer el parser

```
static NSString * const kDataElementName = @"data";
static NSString * const kDataIdElementName = @"dataId";
static NSString * const kHeaderImageFileElementName = @"headerImageFile";
static NSString * const kHeaderTextElementName = @"headerText";
```

Método de Lectura de apertura de etiqueta

Este método se va a encargar de crear los objetos necesarios cuando se lea un comienzo de etiqueta. Dependiendo de la etiqueta que haya leído va a inicializar un tipo de objeto u otro, por ejemplo si lee la correspondiente a NextLevel, inicializará el objeto de NextLevel que estará preparado para tomar el valor cuando se detecte fin de etiqueta

```
(void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName attributes:(NSDictionary *)attributeDict{

  if ([elementName isEqualToString:kDataElementName]) {
      EjemploLevelData *theItem = [[EjemploLevelData alloc] init];
      self.parsedItem = theItem;
      [theItem release];
  }else if ([elementName isEqualToString:kPositionElementName]) {
      EjemploItem* thePositionItem = [[EjemploItem alloc] init];
      self.currEjemploItem = theEjemploItem;
      [thePositionItem release];
  } else if ([elementName isEqualToString:kNextLevelElementName]) {
      NextLevel* theNextLevel = [[NextLevel alloc] init];
      self.currNextLevel = theNextLevel;
      [theNextLevel release];
```

Método de Lectura de cierre de etiqueta

Este método se va a encargar de darle valor a los atributos que hemos creado en el método anterior. Cuando un cierre de etiqueta se detecta, se llama a este método y dependiendo de la etiqueta que haya reconocido, va a almacenar el valor que contenga en un atributo u otro.

Bien una vez llegados a este punto, una vez que ya sabemos la estructura de los parsers y como funcionan vamos a pasar por cada parser explicando de manera general su comportamiento:

NwLevelParser

Estamos ante el padre de todos los parsers encargados parsear xml que definen pantallas exceptuando la pantalla de portada (como podemos ver en el diagrama anterior) y los xml que definen en menú superior e inferior que no son considerados pantallas en si mismos, si no complemento de estas.

Este parser, no se encarga de parsear ningún xml, su misión es definir unos atributos a las que tendrán acceso todos sus hijos. Estos atributos son los siguientes:

- **AppLevelData** : define tanto un mapa como una lista de items que serán los niveles de nuestra aplicación
- **DataItem** : almacena los datos principales de los xml, datos que todos los xml tienen que pueden ser el dataId, headerImageFile, headerText ...

A partir de ahora, todos los parser que vamos a ver, hereden o no de NwLevelParser van a están asociados a un xml.

AppParser

¿Que xml parsea?	Parsea el xml más importante del framework app.xml
¿Que objetos maneja?	* ApplicationData, almacena los datos de la aplicación * AppLevel, define el tipo de pantalla que tenemos
Explicación general	Inicializa y da valor a todos los campos pertenecientes a ApplicationData y AppLevel Su misión es almacenar los datos correctamente para que la navegación a través del framework sea correcta y posible

CoverParser

¿Que xml parsea?	Parsea el xml de la portada cover.xml
¿Que objetos maneja?	*Cover, almacena los datos necesarios para definir la portada *AppButton, almacena los datos necesarios para definir un botón *NextLevel, almacena los NextLevel correspondientes
Explicación general	Inicializa y da los valores a los objetos antes nombrados. Su misión es que podamos acceder a ellos para crear el aspecto y la lógica de la pantalla.

BottomMenuParser

¿Que xml parsea?	Parsea el xml que define el menú inferior, bottom_menu.xml
¿Que objetos maneja?	* BottonMenuData, almacena los datos necesarios para definir el menú * Cover, almacena los datos necesarios para definir la portada * AppButton, almacena los datos necesarios para definir un botón * NextLevel, almacena el nextLevel para poder navegar entre pantallas.
Explicación general	Inicializa y da valor a los objetos antes nombrados. Su misión será crear un menú inferior con las opciones que hayamos fijado desde el panel de control

TopMenuParser

¿Que xml parsea?	Parsea el xml que define el menú superior, top_menu.xml
¿Que objetos maneja?	* TopMenuData, almacena los datos necesarios para definir el menú * Cover, almacena los datos necesarios para definir la portada * AppButton, almacena los datos necesarios para definir un botón * NextLevel, almacena el nextLevel para poder navegar entre pantallas.
Explicación general	Inicializa y da valor a los objetos antes nombrados. Su misión será crear un menú superior con las opciones que hayamos fijado desde el panel de control.

CalendarParser

¿Que xml parsea?	Parsea el xml de la pantalla de calendario
¿Que objetos maneja?	*AppEvents, almacena los datos necesarios para definir un evento * NextLevel, almacena el nextLevel para poder navegar entre pantallas. *CalendarLevelData, contiene un array donde ir almacenando los eventos que vamos parseando
Explicación general	Inicializa y da valor a los objetos antes nombrados y añade a la lista de CalendarLevelData los nuevos eventos creados. Su misión es guardar los eventos creados desde el panel de control para que el calendario pueda mostrarlos

ButtonsParser

¿Que xml parsea?	Parsea el xml de los botones buttons.xml
¿Que objetos maneja?	*AppButtons, almacena los datos necesarios para definir un boton * NextLevel, almacena el NextLevel para poder navegar entre pantallas. *ButtonsLevelData, contiene un array donde ir almacenando los botones que vamos parseando
Explicación general	Inicializa y da valor a los objetos antes nombrados y añade a la lista de ButtonsLevelData los nuevos botones creados. Su misión es poder crear una botonera teniendo en cuenta tanto el aspecto de los botones como su funcionalidad

ImageGalleryParser

¿Que xml parsea?	Parsea el xml de la pantalla de Galería de Imágenes
¿Que objetos maneja?	*ImageGalleryItem, almacena los datos necesarios para definir un GaleryItem * NextLevel, almacena el nextLevel para poder navegar entre pantallas. *ImageGalleryLevelData, contiene un array donde ir almacenando los GaleryItem que vamos parseando
Explicación general	Inicializa y da valor a los objetos antes nombrados y añade a la lista de ImageGalleryLevelData los nuevos GaleryItems creados. Su misión es almacenar las imágenes que y demás propiedades que definen una galería para poder formar tanto su aspecto como su funcionamiento

ImageZoomParser

¿Que xml parsea?	Parsea el xml de la pantalla de Imagen con Zoom
¿Que objetos maneja?	*ImageZoomLevelData, contiene los datos necesarios para definir la pantalla de imagen con zoom
Explicación general	Inicializa y da valor al objeto antes nombrado. Su misión es almacenar los datos necesarios para definir la pantalla de imágen con zoom.

VideoParser

¿Que xml parsea?	Parsea el xml de la pantalla de reproducción de video
¿Que objetos maneja?	*VideoLevelData, almacena los datos necesarios para definir la pantalla de video
Explicación general	Inicializa y da valor al objeto antes nombrado. Su misión es almacenar los datos necesarios para definir la pantalla de video para después poder acceder a ellos y poder reproducirlo

QRParser

¿Que xml parsea?	Parsea el xml de la pantalla de lectura de código QR
¿Que objetos maneja?	*QRLevelData, almacena los datos necesarios para definir la pantalla de lector de códigos QR
Explicación general	Inicializa y da valor al objeto antes nombrado. Su misión es almacenar los datos necesarios para poder definir la pantalla del lector de códigos QR

PdfParser

¿Que xml parsea?	Parsea el xml de la pantalla de lector de Pdf
¿Que objetos maneja?	*PdfLevelData, almacena los datos necesarios para definir la pantalla de lector de pdf
Explicación general	Inicializa y da valor al objeto antes nombrado. Su misión es almacenar los datos necesarios para definir la pantalla de pdf para después poder acceder a la ruta del pdf y poder mostrarlo

WebParser

¿Que xml parsea?	Parsea el xml de la pantalla de contenedor Web
¿Que objetos maneja?	*WebLevelData, almacena los datos necesarios para definir la pantalla del contenedor web
Explicación general	Inicializa y da valor al objeto antes nombrado. Su misión es almacenar los datos necesarios para definir la pantalla del contenedor web para después poder acceder a la ruta fijada desde el panel de control y cargar bien el web site o el html local en el contenedor

MapParser

¿Que xml parsea?	Parsea el xml de la pantalla de mapas
¿Que objetos maneja?	*MapItem, almacena los datos necesarios para definir una marca del mapa *NextLevel, almacena el nextLevel para poder navegar entre pantallas. *MapLevelData, contiene un array donde ir almacenando las marcas del mapa que vamos parseando
Explicación general	Inicializa y da valor a los objetos antes nombrados y añade a la lista de MapLevelData los nuevos MapItems creados. Su misión es almacenar las marcas de mapa y demás propiedades que definen una pantalla de mapa

ListParser

¿Que xml parsea?	Parsea el xml de la pantalla que muestra una lista
¿Que objetos maneja?	*ListItem, almacena los datos necesarios para definir una marca del mapa * NextLevel, almacena el NextLevel para poder navegar entre pantallas. *ListLevelData, contiene un array donde ir almacenando los listItems que vamos parseando
Explicación general	Inicializa y da valor a los objetos antes nombrados y añade a la lista de ListLevelData los nuevos ListItems creados. Su misión es almacenar los ListItems y en el caso de que la lista muestre además imágenes en las celdas también hemos de guardarlas para después poder darle el aspecto deseado

ImageListParser

¿Que xml parsea?	Parsea el xml de la pantalla de Lista + Texto
¿Que objetos maneja?	*ListItem, almacena los datos necesarios para definir una marca del mapa * NextLevel, almacena el NextLevel para poder navegar entre pantallas. *ImageListLevelData, contiene un array donde ir almacenando los listItems que vamos parseando
Explicación general	Inicializa y da valor a los objetos antes nombrados y añade a la lista de ImageListLevelData los nuevos ListItems creados. Su misión es almacenar los ListItems y además almacenar la imagen (en el caso de que la hayamos definido en el panel de control) que va a acompañar a la lista

ImageDescriptionParser

¿Que xml parsea?	Parsea el xml de la pantalla de Imágen + Texto
¿Que objetos maneja?	* NextLevel, almacena el NextLevel para poder navegar entre pantallas. *ImageTextDescripcionLevelData, contiene los datos necesarios para definir una pantalla de Imagen + Texto
Explicación general	Inicializa y da valor a los objetos antes nombrados. Su misión es almacenar los datos necesarios para dotar del comportamiento adecuado y del aspecto requerido a la pantalla de Imagen + Texto

FormParser

¿Que xml parsea?	Parsea el xml de la pantalla de Formulario
¿Que objetos maneja?	* NwAppField, contiene los tipos de campos que se pueden crear así como los atributos necesarios para hacerlo * NextLevel, almacena el NextLevel para poder navegar entre pantallas. *FormLevelData, contiene los datos necesarios para definir una pantalla de Formulario
Explicación general	Inicializa y da valor a los objetos antes nombrados. Su misión es almacenar los datos necesarios para poder crear diferentes campos de manera dinámica

La Importancia de eMobcView

Como bien sabemos, toda pantalla en IOS necesita de un controlador para poder funcionar, ya que en este están presentes todos los IBOutlet y las IBAccion necesarias para poder interactuar con los componentes del nib.

Si hemos echado un corto vistazo a los archivos que contiene el framework, nos habremos dado cuenta de que hay un controlador por cada pantalla, y que además existen un controlador eMobcViewController y un delegado eMobcViewDelegate que parecen ser los principales del framework, pues bien no es que solamente lo parezca, si no que lo son, vamos a explicar esto:

Vamos a empezar con el **eMobcViewDelegate**, y además empezaremos viendo que objetos contiene declarados

```
//
//  eMobcAppDelegate.h
//  eMobc
//
//  Created by Jorge Villaverde on 4/26/11.
//  Copyright Neurowork Consulting SL 2012. All rights reserved.
//

#import <UIKit/UIKit.h>

@class eMobcViewController;

@interface eMobcAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    eMobcViewController *viewController;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet eMobcViewController *viewController;

@end
```

UIWindow	Será la base que contendrá la pila de vistas de nuestra aplicación
eMobcViewController	Necesario para enlazar de alguna manera la UIWindow, base de nuestra pila de Views y además enlazarla con la vista de nuestro controlador.

La funcionalidad del delegate no va mucho más allá de la necesaria simplemente por el sistema de IOS. Los únicos métodos que implementa son los básicos que Apple de pide para que un delegado funcione como tal.

Visto ya el porqué de la existencia del delegado, vamos a centrarnos en el **eMobcViewController** que es quien realmente maneja todo el framework.

Antes de nada vamos a centrarnos en unas capturas para ver que instancias maneja y la estructura de esta clase:

```
@interface eMobcViewController : UIViewController {

//Objects
    NwCoverController *coverController;
    NwController *currentController;

    NwSideMenuController *currentMenuController;

    NSMutableArray* levelsStack;
    NextLevel* currentNextLevel;

//We use it to play the start audio while the cover is charging
    AVAudioPlayer  *player;

//Outlets
    IBOutlet UIView* modelView;

//will have all application data
    ApplicationData* appData;

}
```

NwCoverController	Necesario para poder cargar la propia portada y el loader
NwController	Almacena el controller la View actual. Estamos ante un controlador especial, podríamos llamarle “controlador de controladores”. Para saber más acerca del NwController ve al capitulo de NwController “controlador de controladores”
NwSideMenuController	Será quien soporte el menú lateral
NSMutableArray	Pila de niveles, donde se irán almacenando las View de la aplicación
NextLevel	Almacena el nextLevel que actualmente se está mostrando al usuario
ApplicationData	Almacena los datos de la aplicación
AVAudioPlayer	Reproducirá un sonido mientras carga la aplicación
UIVIew	La vista del eMobcViewController

Una vez que ya hemos pasado por todos los objetos que usa, vamos a ver la estructura que sigue y los métodos más importantes de entender.

Empecemos por los métodos que se encargan de cargar y mostrar el NextLevel (si quieres saber más acerca de los NextLevel, ve al capítulo de NextLevel)

```
- (void) loadNextLevel:(NextLevel*) nextLevel;

//Show next Level but it doesn't stack it in
-(bool) showNextLevel:(NextLevel*) nextLevel;
```

Estos métodos son los encargados de mostrar y cargar los NextLevel.

Por una lado *loadNextLevel* se encarga de apilar el nextLevel que le entra por cabecera y además una vez apilado, llama a *showNextLevel* que es quien realmente se encarga de mostrar al usuario la vista adecuada.

```
/**
 * Load the Level (NextLevel).
 *
 * @param nextLevel level square with each type of View
 */
- (void) loadNextLevel:(NextLevel*) nextLevel{
    if (nextLevel == nil) {
        return;
    }

    if([self showNextLevel:nextLevel]){
        // Add NextLevel to the Stack
        [levelsStack addObject:nextLevel];
    }
}
```

Pero la pregunta viene cuando decimos ...

¿Cómo sabe que NextLevel tiene que cargar? , ¿Que tipo de pantalla tiene que mostrar?

Para ello necesitamos conocer el tipo de actividad que queremos mostrar, que hemos leído previamente del app.xml, pero aún así no sabemos como llegar a los datos para cargar el xml y el tipo de pantalla adecuados.

Para ello tenemos que tomar el AppLevel que guarda el type (tipo de la pantalla) mediante el levelId o el levelNumber del NextLevel, todo eso lo hacemos como se muestra en el código que tenemos a continuación.

```
[. . .]

AppLevel* appLevel = nil;

    if(nextLevel.levelId != nil){
        appLevel = [appData getLevelById:nextLevel.levelId];
    }else {
        appLevel = [appData getLevelByNumber:nextLevel.levelNumber];
    }


    if (appLevel != nil) {
        LoadUtil* util = nil;
        util = [[LoadUtil alloc] initWithValues:appLevel nextLevel:nextLevel];

        SEL loadSelector = nil;

        switch (appLevel.type) {
            case IMAGE_TEXT_DESCRIPTION_ACTIVITY:
                loadSelector = @selector(loadImageTextNextLevel:);
                break;
            case IMAGE_LIST_ACTIVITY:
                loadSelector = @selector(loadImageListNextLevel:);
                break;
[. . .]
```

La relación entre LevelId y lDataId queda explicada en el capítulo dedicado a los NextLevel, por lo que si quieres saber más acerca de estas relaciones ve al capítulo de NextLevel.

Una vez vista la manera en la que el eMobcViewController (controlador principal) carga y muestra los NextLevel, vamos a centrarnos en como llama a los diferentes nextLevel dependiendo del tipo de pantalla y como invoca a su controlador.

Para ello hemos capturado varias porciones de código sobre las que vamos a ir explicándolo.

```
//call nextLevel of each view
    -(void) loadImageTextNextLevel:(LoadUtil*) util;
    -(void) loadImageListNextLevel:(LoadUtil*) util;
    -(void) loadListNextLevel:(LoadUtil*) util;
    -(void) loadImageZoomNextLevel:(LoadUtil*) util;
    -(void) loadImageGalleryNextLevel:(LoadUtil*) util;
    -(void) loadButtonsNextLevel:(LoadUtil*) util;
    -(void) loadFormNextLevel:(LoadUtil*) util;
    -(void) loadMapNextLevel:(LoadUtil*) util;
    -(void) loadVideoNextLevel:(LoadUtil*) util;
    -(void) loadWebNextLevel:(LoadUtil*) util;
    -(void) loadPdfNextLevel:(LoadUtil*) util;
    -(void) loadQRNextLevel:(LoadUtil*) util;
    -(void) loadCalendarNextLevel:(LoadUtil*) util;
```

Todos estos métodos son invocados desde showNextLevel y cargan el nextLevel de cada tipo de actividad que soporta el framework, ¿Cómo lo hacen? lo vemos a continuación:

```
/**
 * Load the level and call to their own controller method
 *
 * @param util
 *
 * @see loadCalendarController
 */
- (void) loadCalendarNextLevel:(LoadUtil*) util{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    //Read calendar.xml and save it into theData (CalendarLevelData type)
    CalendarLevelData* theData = [[NwUtil instance] readCalendarData:util.appLevel
                                                           nextLevel:util.nextLevel];

    //Charge data into controlerUtil
    LoadControlerUtil* controlerUtil = nil;
    controlerUtil = [[LoadControlerUtil alloc] initWithValues:util.appLevel
                                                     theData:theData];

    //loadCalendarController and we init its data with controlerUtil data
    [self performSelectorOnMainThread:@selector(loadCalendarController:)
                           withObject:controlerUtil
                        waitUntilDone:NO];

    [controlerUtil release];
    [pool drain];
}
```

La manera en la que cargamos el nextLevel indicando el controlador es bastante fácil;

1. Lo primero que hacemos es cargar los datos que están presentes en el xml de cada pantalla (calendar.xml en este caso).

   Para ello utilizamos un método de ***NwUtils*** que se encarga de llamar al parser y a su xml asociado.

   (para saber más sobre el funcionamiento de esta clase ve al capítulo de *NwUtils*)
2. Almacenamos esos datos en una instancia de tipo ***LoadControllerUtil***. Lo inicializamos con el appLevel asociado y le metemos los datos que acabamos de obtener del xml.
3. Por último, llamamos a ***loadCalendarController***, encargado de llamar al controlador de esta vista y le inicializamos con los datos que previamente hemos cargado en *LoadControllerUtil*.

Llegados a este punto, ya tenemos:

- Los datos parseados del xml
- El controlador inicializado con los datos que hemos parseado

Por lo que sólo nos queda el último paso para cargar la pantalla que es explicar el funcionamiento del método que se encarga de cargar el controlador recién inicializado:

```
/**
 * Load their own controller method
 *
 * @param theControlerUtil
 *
 * @see loadController
 */
- (void) loadCalendarController:(LoadControlerUtil*)  theControlerUtil{
    //Get nib name
    NSString* controllerNibName = [eMobcViewController addIPadSuffixWhenOnIPad:@"NwCalendarController"];
    //Init controller (calendarController) with a especific nib name
    NwCalendarController *controller = [[NwCalendarController alloc] initWithNibName:controllerNibName
                                                                               bundle:nil];

    controller.data = (CalendarLevelData*)theControlerUtil.data;
    controller.mainController = self;

    //Asing the NwController (currentController) the new controller of the new view
    self.currentController = controller;

    //load and show controller whit a transition
    [self loadController:currentController withAppLevel:theControlerUtil.appLevel];
    [controller release];
}
```

Para cargar el nuevo controlador seguimos los siguientes pasos:

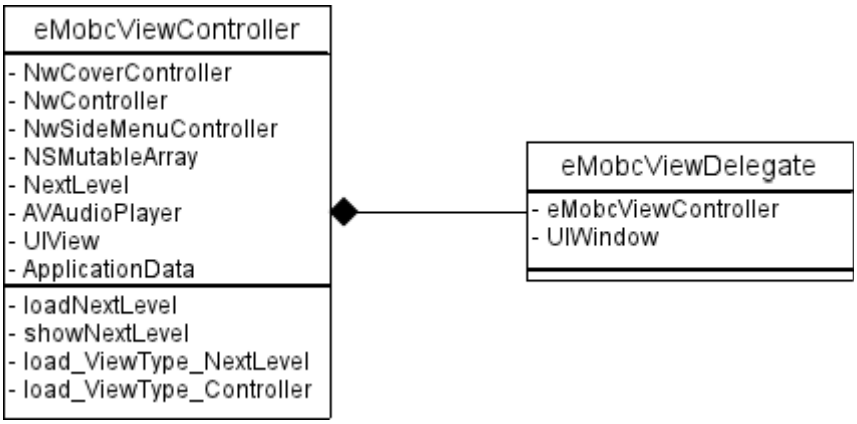
1. Tomamos el nombre del nib concatenando (en el caso de que estemos ante un iPad) el sufijo -iPad
2. Creamos una nueva instancia del controlador deseado (en este caso *NwCalendarController*) y le inicializamos con el nombre del nib que acabamos de obtener
3. Asignamos los datos que nos han pasado desde *loadCalendarNextLevel* a la instancia de *CalendarLevelData* que contiene el propio *NwCalendarController*
4. Asignamos al currentController (*NwController*) el nuevo controlador que acabamos de declarar
5. Por último llamamos al método *loadController* para que nos cargue el nuevo controlador que ya contiene todos los datos y tiene asociado el nib.

Una vez llegados hasta aquí ya sabemos como funciona el eMobcViewController y como llama a los diferentes controladores, carga los diferentes niveles...

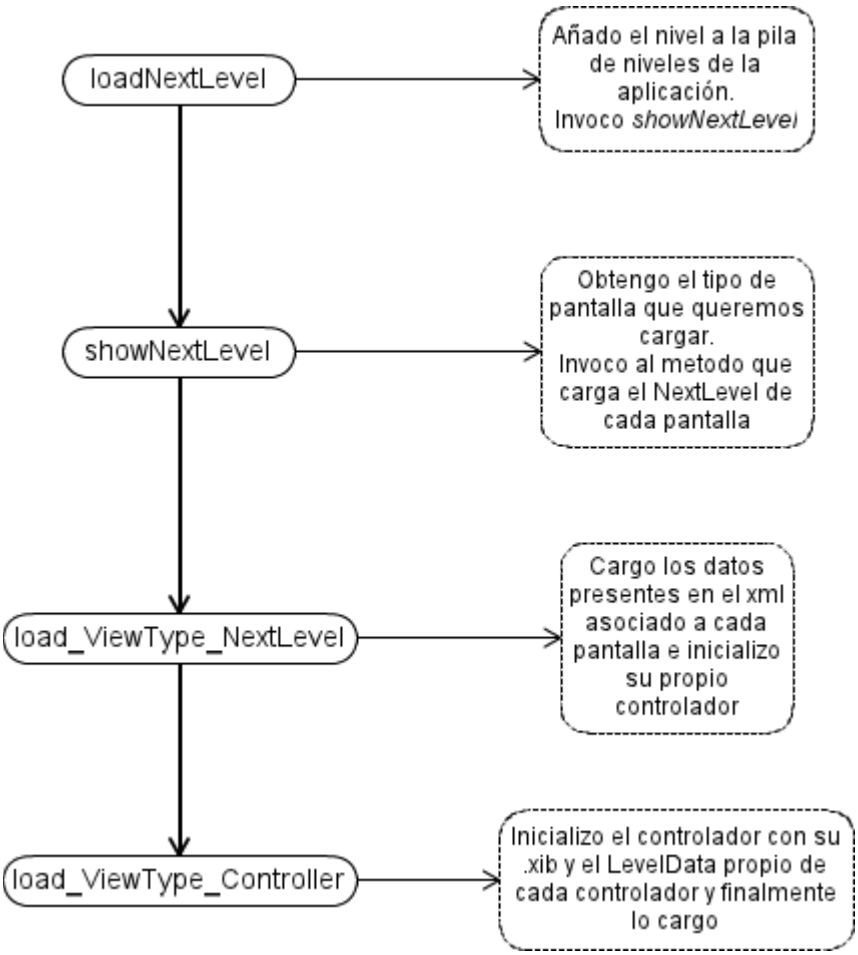
Ya sabemos la importancia del eMobcViewController.

Para que todo quede lo más claro posible hemos adjuntado un diagrama que me indica el flujo de ficheros y métodos que hemos seguido dentro de esta explicación.

Diagrama de clases que muestra los atributos y métodos mencionados:



Flujo que se sigue al invocar un nextLevel específico (anteriormente explicado):



La Importancia de NwUtil

NwUtil tiene una gran misión básica que es asociar el parser adecuado al xml adecuado. Para ello cuenta con una serie de atributos y una serie de métodos que vamos a ver a continuación con un poco de detalle.

Empecemos con los atributos:

```
@interface NwUtil : NSObject {

//Objects
    ApplicationData *theAppData;
    Cover* theCover;
    TopMenuData* theTopMenu;
    BottomMenuData* theBottomMenu;
}
```

ApplicationData	Guarda los datos presentes en app.xml mediante AppParser
Cover	Guarda los datos presentes en cover.xml mediante CoverParser
TopMenuData	Guarda los datos presentes en top_menu.xml mediante TopMenuParser
BottonMenuData	Guarda los datos presentes en botton_menu.xml mediante BottomMenuParser

Si observamos los atributos que hemos declarado y si nos hemos leído el capítulo dedicado a los parser (si no lo has hecho aún, te recomendamos que le eches un vistazo) nos daremos cuenta que sólo tenemos instancias de aquellos archivos cuyos parser eran especiales, no heredan de NwLevelParser, puesto que van a recibir un trato ligeramente diferente.

Empecemos viendo los métodos que invocaran al parser especial para los atributos que hemos mencionado.

```
-(ApplicationData *) readApplicationData;
-(Cover*) readCover;
-(TopMenuData*) readTopMenu;
-(BottomMenuData*) readBottomMenu;
-(AppLevel*) readAppLevel:(NextLevel*)nextLevel;
-(DataItem*) readAppLevelData:(NextLevel*)nextLevel;
```

```
/**
 * Read app.xml file
 *
 * @return Return datas saved in theAppData
 */
-(ApplicationData *) readApplicationData {

    if (theAppData == nil) {
        //Get app.xml path
        NSString *path = [[[NSBundle mainBundle] resourcePath] stringByAppendingPathComponent:@"app.xml"];

        //Indicate to AppParser where app.xml is
        AppParser* appParser = [[AppParser alloc] init];
        [appParser parseXMLFileAtURL:path];

        //Asing datas from AppParser to ApplicationData
        theAppData = appParser.currentApplicationDataObject;
        [appParser release];
    }

    return theAppData;
}
```

Bien, expliquemos el funcionamiento de estos métodos, he preferido explicar sólo uno de ellos ya que su funcionamiento va a ser similar.

Expliquemos su funcionamiento:

1. Lo primero que hacemos es crear un NSString (path) que va a ser la dirección path del string dado, que en este caso se corresponde con el nombre de app.xml
2. Creamos un parser de tipo AppPaser (parser asociado a app.xml) y le introducimos la path que acabamos que crear para que tome el xml correcto
3. Finalmente indicamos que los datos que ha recogido el AppParser se corresponden con los datos de ApplicationData (theAppData)

Una vez llegados hasta aquí, sólo nos queda saber como invocamos al parser adecuado y que además este lea el xml que le corresponde. Este método va tener una estructura que van a seguir todos aquellas pantallas cuyos parser hereden de NwLevelParser

```
-(ImageTextDescriptionLevelData*) readImageTextDescriptionData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
-(ImageListLevelData*) readImageListData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
-(ListLevelData*) readListData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
-(ImageZoomLevelData*) readImageZoomData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
-(ImageGalleryLevelData*) readImageGalleryData:(LoadUtil*) util;
-(ButtonsLevelData*) readButtonsData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
-(FormLevelData*) readFormData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
-(MapLevelData*) readMapData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
-(VideoLevelData*) readVideoData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
-(WebLevelData*) readWebData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
-(PdfLevelData*) readPdfData:(AppLevel *)appLevel nextLevel:(NextLevel *)nextLevel;
-(QRLevelData*) readQRData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
-(CalendarLevelData*) readCalendarData:(AppLevel *)appLevel nextLevel:(NextLevel *)nextLevel;
```

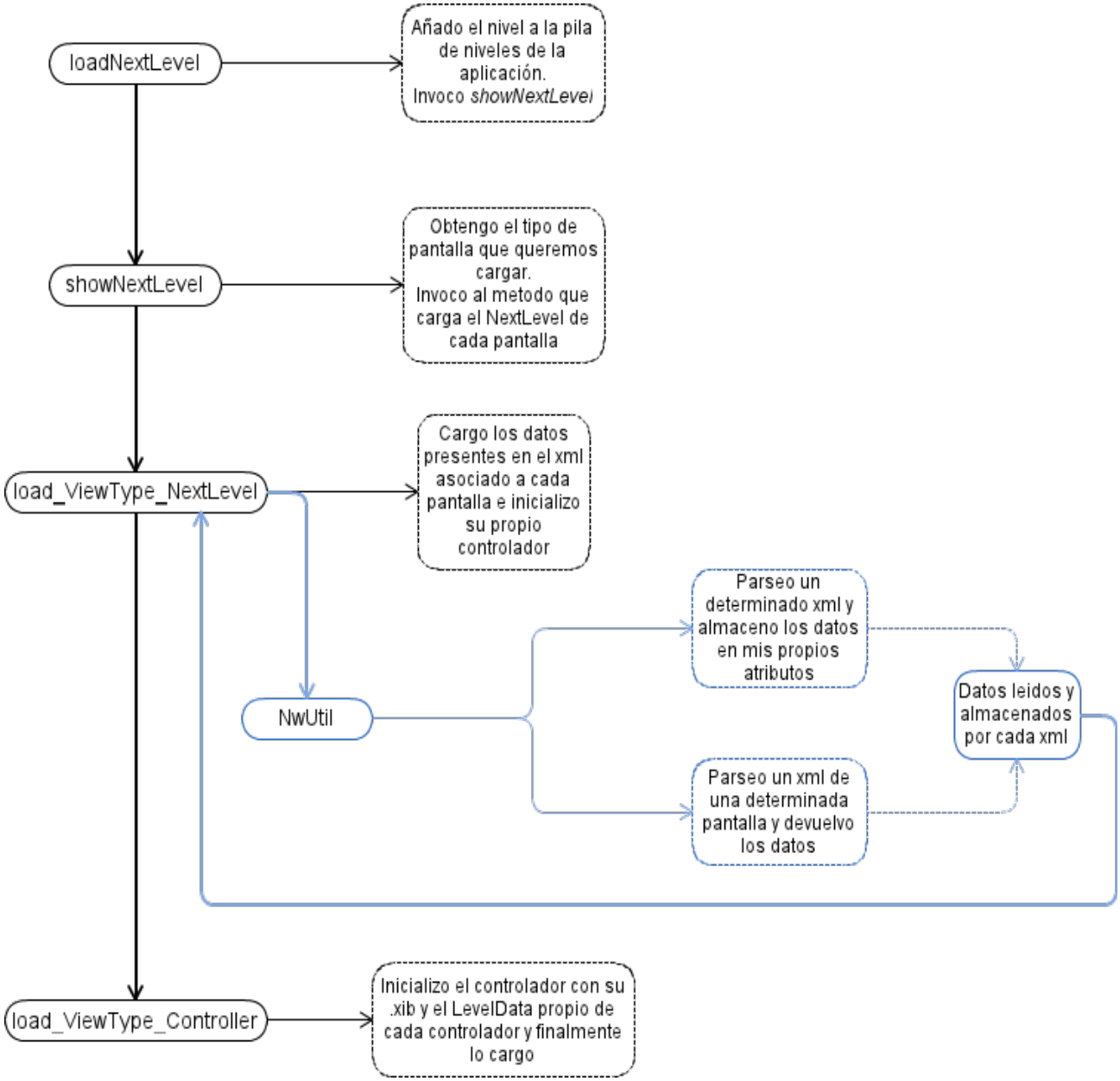
Debido a que todos ellos van a seguir la misma estructura, vamos a explicar sólo uno de ellos:

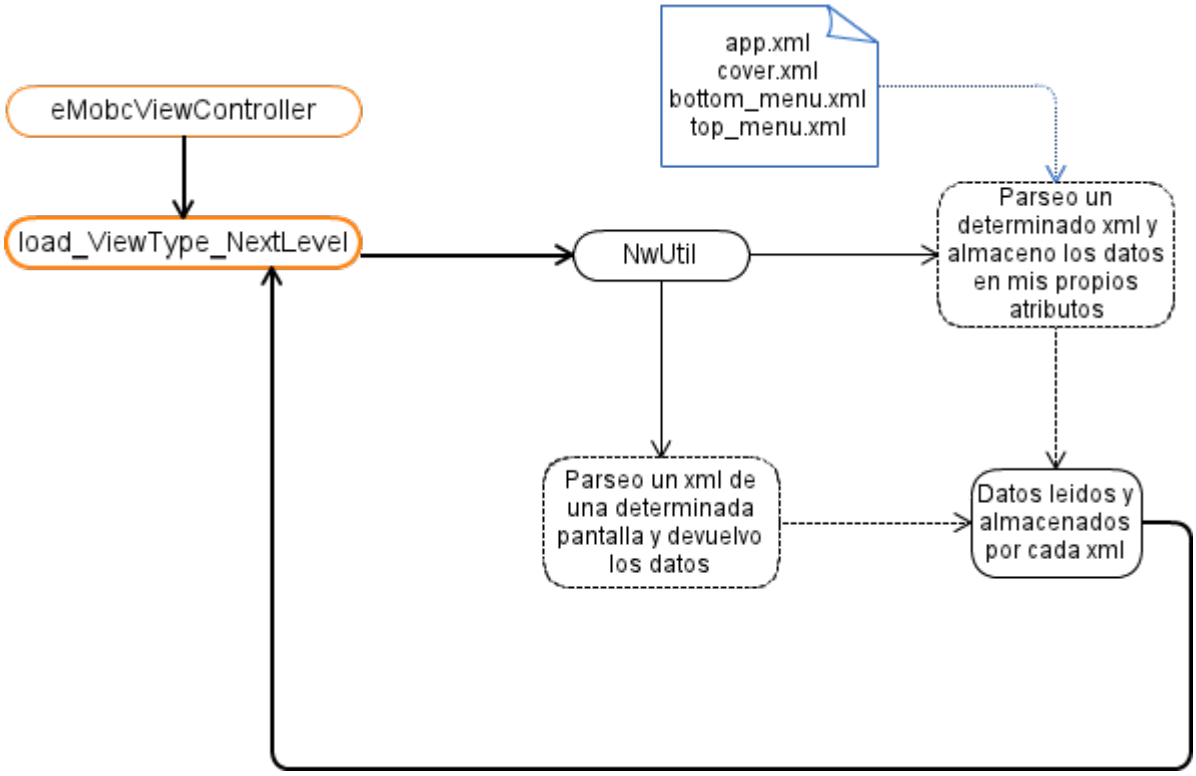
```
/**
 * Read date from calendar.xml
 *
 * @param appLevel Application Level
 * @param nextLevel The next level where we can go thought this View
 *
 * @return Return datas saves in CalendarLevelData
 *
 * @see dataItemById
 * @see dataItemByNumber
 */
-(CalendarLevelData*) readCalendarData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel {
    //Create CalendarParser
    CalendarParser *parser = [[CalendarParser alloc] init];

    //Get path from calendar.xml
    NSData* parseData = [appLevel.file content];
    //Indicate to parser which xml have to parse
    [parser parseXMLFileFromData:parseData];

    //Asing data from CalendarParser to CalendarLevel
    CalendarLevel* theLevel = (CalendarLevel*)parser.parsedLevel;

    //Get the NextLevel we want to show (look for it into itemsMap)
    DataItem* theItem = nil;
    if (nextLevel.dataId != nil) {
        theItem = [theLevel dataItemById:nextLevel.dataId];
    }else {
        theItem = [theLevel dataItemByNumber:nextLevel.dataNumber];
    }

    [parser release];
    return (CalendarLevelData*)theItem;
}
```

Expliquemos los pasos que seguimos cuando queremos parsear un xml asociado a un tipo de pantalla:

1. Lo primero que hacemos es inicializar el parser específico para cada tipo de pantalla, en nuestro caso CalendarParser
2. Para indicarle al parser que hemos creado que xml tiene que leer, necesitamos tomar la path del xml. Para ello accedemos al file de AppLevel y se lo indicamos posteriormente al parser
3. Por último devolvemos el nextLevel de la pantalla a la que queremos acceder, la que estamos leyendo actualmente

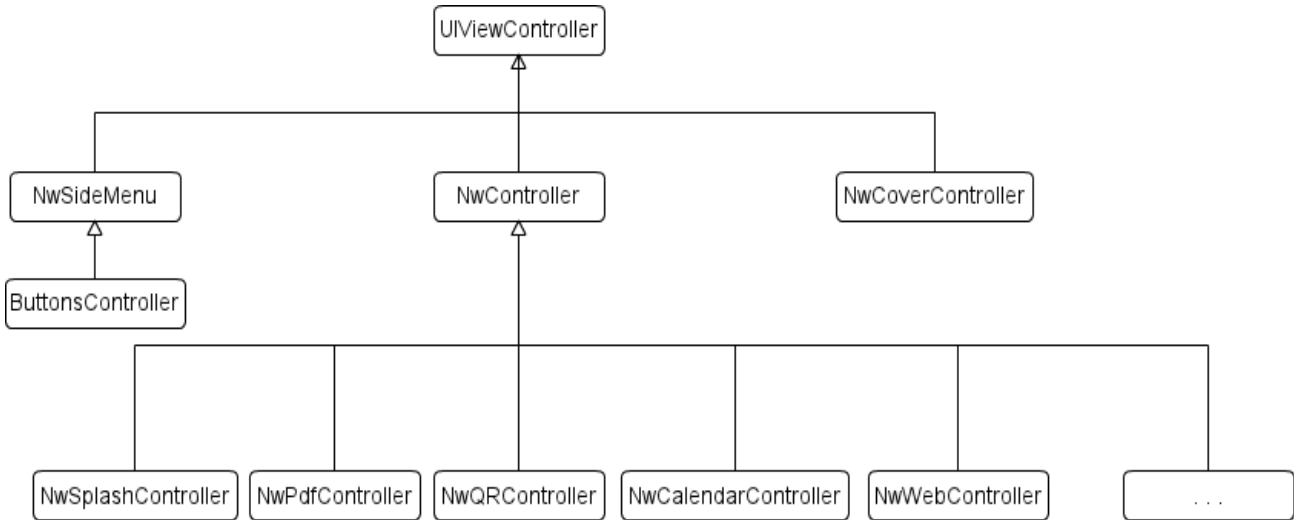
Para que la explicación quede lo más clara posible vamos a acompañar la explicación con un esquema. Además debido a que esta parte tiene una estrecha relación con el capítulo de “La importancia de eMobcView”, vamos a ampliar el diagrama antes expuesto :




app.xml
cover.xml
bottom_menu.xml
top_menu.xml
eMobcViewController
load_ViewType_NextLevel
NwUtil
Parseo un determinado xml y almaceno los datos en mis propios atributos
Parseo un xml de una determinada pantalla y devuelvo los datos
Datos leidos y almacenados por cada xml

La Importancia del NwController

Podríamos decir que el NwController es el controlador de los controladores, porque todos los controladores (exceptuando NwSideMenuController y CoverController) heredan de NwController, permitiendo así tener acceso a todo lo que este contenga, como indica el diagrama que tenemos a continuación.



Antes de empezar a explicar la razón de que exista esta relación vamos a ver qué atributos tiene para que podamos empezar a encajar las piezas del puzzle

```
//Objects
     [. . .]

    eMobcViewController* mainController;

     [. . .]
```

Como podemos ver, tenemos una instancia de eMobcViewController lo que nos permite tener acceso a todos los métodos y atributos (que en este caso es lo que más nos interesa) del eMobcViewController.

(si quieres saber más sobre el eMobcViewController ve al capítulo de 'La importancia del eMobcView')

¿Pero por qué existe esta relación de herencia?

La respuesta puede empezar a ser evidente.

Ahora mismo tenemos que ser prácticos, y siempre pensar en como hacer para que nuestro código sea lo más económico posible, es decir, qué pasos podemos seguir para poder reutilizar código.

Como bien sabemos hay ciertas características y funcionalidades que todas las pantallas van a compartir como los banners de publicidad (si quieres saber un poco más, ve al apartado de 'un pequeño inciso para la publicidad'), los menús inferiores o superiores, los estilos... es decir, todo aquello que puede ser común a las demás pantallas y que puede ser tratado de la misma manera se incluirá dentro de este fichero.

(Si quieres saber más acerca de menús o estilos, ve a los capítulos de 'Funcionamiento de Menús' y 'Funcionamiento de estilos' respectivamente)

De esta manera si como desarrolladores queremos incluir una característica que va a poder estar presente en todas (o casi todas) las demás pantallas deberemos incluirla dentro de este controlador para evitar repetir código innecesariamente

Si dejamos a un lado los métodos que pueda contener NwController, nos podemos centrar en la instancia de eMobcViewController (si quieres saber más sobre el eMobcViewController ve al capítulo de 'La importancia del eMobcView').

Contener precisamente una instancia de eMobcViewController me va a permitir acceder a sus atributos y métodos no sólo desde el NwController si no desde todas las clases que heredan de él permitiendo así poder llegar desde cualquier controlador a los datos del controlador principal (mainController).

Pongamos un ejemplo de estas dos funcionalidades aprovechando para hacer un...

Pequeño Inciso para la Publicidad

El framework soporta dos tipos de publicidad

- Publicidad de adMob
- Publicidad de Yoc

Para que se puedan mostrar los banners hemos utilizados dos librerías diferentes que ya contenían toda la lógica para poder utilizarlos, estas librerías son:

- libGoogleAdMobAds específica para la publicidad de adMob
- SmartAdServer específica para la publicidad de Yoc

```
//publicity
    GADBannerView * admobBanner_; //admob
    SmartAdServerView *yocBanner_; //yoc
```

¿Pero qué tiene esto que ver con la importancia del NwController?

Tiene que ver con la importancia desde dos puntos de vista diferentes

- Por un lado, queremos que todas las pantallas contengan (en el caso de que nuestra aplicación lleve publicidad) banners publicitarios y que además estos sean de la misma empresa y estén en la misma posición de la pantalla.

Esto es una característica común a cada pantalla, por lo que en lugar de repetir el código en cada uno de los controladores bastará con incluirlo dentro del padre de todos ellos para que la publicidad esté accesible desde todos ellos

- Por otro lado, la posibilidad de que nuestra aplicación contenga banner así como el tipo y la posición del mismo, queda indicado dentro del app.xml y almacenado dentro de ApplicationData. Para saber si tenemos que cargar el banner así como sus características tenemos que acceder a esos datos y como ya hemos visto (ve al capítulo de eMobCView) el controlador principal eMobcViewController contiene una instancia de ApplicationData ya que es quién necesita esos datos.

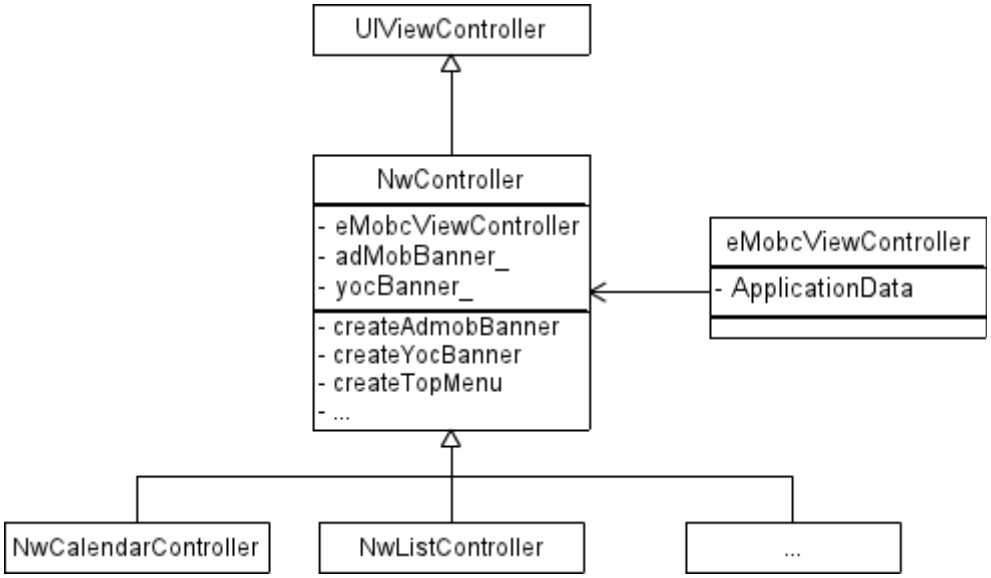
Gracias a la relación composición que tiene NwController podemos acceder a esos datos sin necesidad de crear una instancia de ApplicationData y volver a parsear el app.xml ya que esto restaría en optimización.

```
-(void)viewDidLoad {

    [. . .]

    //publicity
    if([mainController.appData.banner isEqualToString:@"admob"]){ //if we have publicity from adMob
        [self createAdmobBanner];
    }else if([mainController.appData.banner isEqualToString:@"yoc"]){ //if we have publicity from Yoc
        [self createYocBanner];
    }

    [. . .]
```

Llegados hasta aquí hemos comprendido la importancia del NwController y de las relaciones de herencia que mantiene con todos los demás controladores.

Siempre tenemos que recordar que buscamos la optimización del código y que una manera de conseguirlo es precisamente juntar todo aquello común todas las pantallas e implementarlo dentro de NwController.

Para que todo quede lo más claro posible vamos a terminar la explicación con un diagrama resumen de la relaciones de herencia y composición



Guía de programación

Cómo crear un nuevo tipo de ventana para el proyecto iOS

El siguiente documento es una guía de programación para crear un nuevo tipo de ventana. Muestra en detalle los archivos y métodos que hay que modificar para conseguir que el framework trabaje con el nuevo tipo de ventana apoyándose en un ejemplo concreto (ventana de calendario).

Puede consultar también la “guía rápida nuevo tipo de ventana”.

Paso 1: Tipo de actividad

La nueva ventana tiene su propio tipo de Activity. Existe un enumerado en el archivo *ActivityType.m.* Este enumerado se utiliza para comprobar, más adelante, durante la ejecución y creación de pantallas, de qué tipo es la actividad que queremos crear.

```
typedef enum ActivityType {
      COVER_ACTIVITY,
      IMAGE_TEXT_DESCRIPTION_ACTIVITY,
      IMAGE_LIST_ACTIVITY,
      LIST_ACTIVITY,
      VIDEO_ACTIVITY,
      IMAGE_ZOOM_ACTIVITY,
      IMAGE_GALLERY_ACTIVITY,
      BUTTONS_ACTIVITY,
      FORM_ACTIVITY,
      MAP_ACTIVITY,
      WEB_ACTIVITY,
      PDF_ACTIVITY,
      QR_ACTIVITY,
      SPLASH_ACTIVITY,
      SEARCH_ACTIVITY,
      CALENDAR_ACTIVITY,
      QUIZ_ACTIVITY,
      CANVAS_ACTIVITY,
      AUDIO_ACTIVITY
}ActivityType;
```

Para añadir un nuevo tipo de ventana es necesario, por tanto, modificar el enumerado para incluir el nuevo valor correspondiente a nuestra nueva Actividad.

La nomenclatura utilizada es *ACTIVITY_NAME*_ACTIVITY. Donde se sustituye ACTIVITY_NAME por el nombre de la actividad. Durante esta guía vamos a crear un nuevo tipo de ventana de calendario. Para este ejemplo crearíamos un nuevo caso dentro de ActivityType.m.

Paso 2: Crear y definir un xml

El nuevo tipo de ventana tiene que estar definido previamente y esto incluye que el panel de control también tiene que saber que existe. Por lo tanto, desde el panel de control se podrá crear esta nueva ventana y se creará un XML con los datos e información necesarios para generarla desde el proyecto Android.

Este archivo XML va a contener los datos para poder crear la ventana. Sin embargo, no tiene porqué ser igual a otros archivos XML. Por ejemplo, una ventana de galería no necesita los mismos datos que una ventana de vídeo. Nuestra nueva ventana puede necesitar una estructura y datos concretos que la diferencian de las demás. Es necesario definir una nueva estructura para esta ventana.

En el caso del Calendario se ha decidido incluir los siguientes datos (ver figura 2):

- **DataId**: para identificar los datos
- **headerImageFile**: Imagen de la cabecera
- **headerText**: Texto de la cabecera
- **events**: Lista de eventos del calendario
- **event**: Evento concreto.

- **title**: Nombre del evento
- **eventDate**: Fecha del evento
- **text**: Descripción del evento
- **NextLevel**: Acción al hacer click

A continuación se muestra un ejemplo de este xml:

```
<levelData>
	<data>
		<dataId>calendar</dataId>
		<headerImageFile>images/white_title.png</headerImageFile>
		<headerText>Información</headerText>
		<events>
			<event>
				<title>Matriculación universidad</title>
				<eventDate>24/07/2012</eventDate>
				<time>23:00</time>
				<text>Consulta el pdf</text>
				<Next Level>
					<nextLevelLevelId>pdf</nextLevelLevelId>
					<nextLevelDataId>pdf</nextLevelDataId>
				</Next Level>
			</event>
		</events>
	</data>
</levelData>
```

Paso 3: Crear datos de la aplicación

Una vez tenemos creada y definida la estructura que tendrán los XML que el panel de control creará cada vez que tengamos una ventana de nuestro tipo, es necesario implementar las clases que definan el tipo de datos en tiempo de ejecución. Estas clases se crearán una vez parseamos el XML y contendrán la misma información. La diferencia es que estas clases serán utilizadas durante la ejecución de la aplicación para adquirir más fácil y rápidamente los datos sin tener que parsear continuamente los XML.

Se pueden crear todas las clases que sean necesarias para almacenar los datos, pero es necesario que exista un *DataItem* que es el que almacenará todo el conjunto de datos necesario para la ventana. Esta clase tiene que heredar de *AppLevelDataItem* para tener atributos básicos necesarios para el framework.

Para el ejemplo del Calendario se han creado las siguientes clases que modelan el tipo de datos que utilizará (eventos):

- *NuevaPantallaLevelData.m*: Es la clase principal que almacena datos de un calendario (lista de Eventos). Extiende de DataItem para tener algunos atributos generales como headerText.
- *EventDataItem.java*: Clase para almacenar los datos relativos a un evento como son título, descripción o fecha.

```
#import <Foundation/Foundation.h>
#import "DataItem.h"

@interface NuevaPantallaLevelData : DataItem {
      NSString* npData1;
      NSString* npData2;
      NSString* npData3;
}

@property (nonatomic, copy) NSString* npData1;
@property (nonatomic, copy) NSString* npData2;
@property (nonatomic, copy) NSString* npData3;

@end
```

```
#import "NuevaPantallaLevel.h"

- (void) addLevelDataItem:(NuevaPantallaLevelData*) newDataItem{
      [super addItem:(DataItem*)newDataItem];
}

@end
```

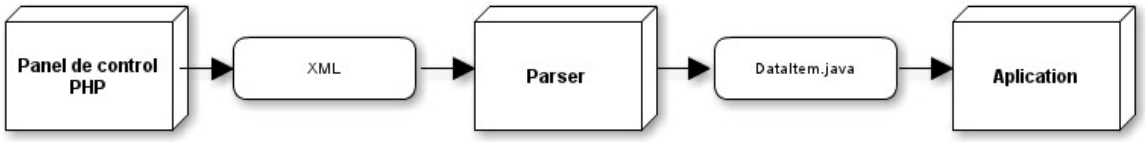
```
-(NuevaPantallaLevelData*) readListData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel {
      NuevaPantallaParser *parser = [[NuevaPantallaParser alloc] init];

      NSData* parseData = [appLevel.file content];
      [parser parseXMLFileFromData:parseData];

      NuevaPantallaLevel* theLevel = (NuevaPantallaLevel*)parser.parsedLevel;

      DataItem* theItem = nil;
      if (nextLevel.dataId != nil) {
        theItem = [theLevel dataItemById:nextLevel.dataId];
      }else {
        theItem = [theLevel dataItemByNumber:nextLevel.dataNumber];
      }

      [parser release];
      return (NuevaPantallaLevelData*)theItem;
}


-(DataItem*) readAppLevelData:(NextLevel*)nextLevel {

      …
```

```
	switch (theLevel.type) {

	  case LIST_ACTIVITY:
		theData = [self readListData:theLevel nextLevel:nextLevel];
		break;
	  case CALENDAR_ACTIVITY:
		theData = [self readCalendarData:theLevel nextLevel:nextLevel];
		break;
	  case QUIZ_ACTIVITY:
		theData = [self readQuizData:theLevel nextLevel:nextLevel];
		break;
	  case CANVAS_ACTIVITY:
		theData = [self readCanvasData:theLevel nextLevel:nextLevel];
		break;
	  case AUDIO_ACTIVITY:
		theData = [self readAudioData:theLevel nextLevel:nextLevel];
		break;
	  case NUEVAPANTALLA_ACTIVITY:
		theData = [self readNuevaPantallaData:theLevel nextLevel:nextLevel];
		break;
	  default:
		break;
	}
	[util release];
	[theLevel release];
	return theData;
}


- (void) loadNuevaPantallaNextLevel:(LoadUtil*) util{
	NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

	NuevaPantallaLevelData* theData = [[NwUtil instance] readNuevaPantallaData:util.appLevel

		nextLevel:util.nextLevel];

	theData.parentNextLevel = self.currentNextLevel;

	LoadControlerUtil* controlerUtil = nil;
	controlerUtil = [[LoadControlerUtil alloc] initWithValues:util.appLevel

theData:theData];

	[util release];

	[self performSelectorOnMainThread:@selector(loadNuevaPantallaController:)
					withObject:controlerUtil
				waitUntilDone:NO];

	[pool drain];
}



- (void) loadNuevaPantallaController:(LoadControlerUtil*)  theControlerUtil{
	NSString* controllerNibName = [eMobcViewController
addIPadSuffixWhenOnIPad:@"NwNuevaPantallaController"];

	NwNuevaPantallaController *controller = [[NwNuevaPantallaController alloc]
initWithNibName:controllerNibName

					bundle:nil];
	controller.data = (NuevaPantallaLevelData*)theControlerUtil.data;
	controller.mainController = self;

	self.currentController = controller;

	[self loadController:currentController withAppLevel:theControlerUtil.appLevel];
	[controller release];
	[theControlerUtil release];
}

- (bool) showNextLevel:(NextLevel*) nextLevel{
```

```
		[…]
		switch (appLevel.type) {
			[…]
			case QUIZ_ACTIVITY:
				loadSelector = @selector(loadQuizNextLevel:);
				break;
			case CANVAS_ACTIVITY:
				loadSelector = @selector(loadCanvasNextLevel:);
				break;
			case AUDIO_ACTIVITY:
				loadSelector = @selector(loadAudioNextLevel:);
				break;
			default:
				break;
		}
	[…]
}
```

Paso 4: Añadir parser

Una vez tenemos definido el XML (paso 2) que contendrá los datos, y las clases que almacenarán estos datos durante la ejecución de la aplicación (paso 3), falta definir el elemento que transformará de uno al otro: el parser.




```
static NSString * const kActTypeCalendar = @"CALENDAR_ACTIVITY";
static NSString * const kActTypeQuiz = @"QUIZ_ACTIVITY";
static NSString * const kActTypeCanvas = @"CANVAS_ACTIVITY";
static NSString * const kActTypeAudio = @"AUDIO_ACTIVITY";
static NSString * const kActTypeNuevaPantalla =@"NUEVAPANTALLA_ACTIVITY";

static NSString * const kDataElementName = @"data";
static NSString * const kDataIdElementName = @"dataId";
static NSString * const kHeaderImageFileElementName = @"headerImageFile";
static NSString * const kHeaderTextElementName = @"headerText";


- (void)parser:(AQXMLParser *)parser didStartElement:(NSString *)elementName namespaceURI:(NSString
*)namespaceURI qualifiedName:(NSString *)qName attributes:(NSDictionary *)attributeDict{

      if([elementName isEqualToString:kDataElementName]){
        EjemploLevelData *theItem = [[EjemploLevelData alloc] init];
        self.parsedItem = theItem;
        [theItem release];
      } else if([elementName isEqualToString:kPositionElementName]){
        EjemploItem* theEjemploItem = [[EjemploItem alloc] init];
        self.currEjemploItem = theEjemploItem;
        [theEjemploItem release];
      } else if([elementName isEqualToString:kNextLevelElementName]){
        NextLevel* theNextLevel = [[NextLevel alloc] init];
        self.currNextLevel = theNextLevel;
        [theNextLevel release];
```

Paso 5: Añadir Activity

```
#import <UIKit/UIKit.h>

@class eMobcViewController;

@interface eMobcAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    eMobcViewController *viewController;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet eMobcViewController *viewController;

@end


@interface eMobcViewController : UIViewController {
```

```
//Objetos
      NwCoverController *coverController;
      NwController *currentController;

      NwSideMenuController *currentMenuController;

      NSMutableArray* levelsStack;
      NextLevel* currentNextLevel;

//We use it to play the start audio while the cover is charging
      AVAudioPlayer  *player;

//Outlets
      IBOutlet UIView* modelView;

//save all data from app.xml
      ApplicationData* appData;

//Format and styles
      StylesLevelData* theStyle;
      FormatsStylesLevelData* theFormat;

}
```

StylesLevelData : Almacena el estilo de la aplicación

FormatStylesLevelData : Almacena los formatos del estilo

```
-(void) loadNextLevel:(NextLevel*) nextLevel;
/*
 Show next Level but it doesn't stack it in
 */
-(bool) showNextLevel:(NextLevel*) nextLevel;



/**
 * Load the Level (NextLevel).
 *
 * @param nextLevel level square with each type of View
 */
- (void) loadNextLevel:(NextLevel*) nextLevel{
      if (nextLevel == nil) {
        return;
      }

      if([self showNextLevel:nextLevel]){
        // Add NextLevel to the Stack
        [levelsStack addObject:nextLevel];
      }
}


[...]
AppLevel* appLevel = nil;

      if(nextLevel.levelId != nil){
        appLevel = [appData getLevelById:nextLevel.levelId];
      }else {
        appLevel = [appData getLevelByNumber:nextLevel.levelNumber];
      }

      SEL loadSelector = nil;

      if([nextLevel.levelId isEqualToString:@"emobc"] && [nextLevel.dataId
isEqualToString:@"profile"]){
        [self loadProfile];
      }else if([nextLevel.levelId isEqualToString:@"emobc"] && [nextLevel.dataId
isEqualToString:@"search"]){
        [self loadSearchNextLevel];
      }else{
```

```
        if (appLevel != nil) {

                LoadUtil* util = nil;
                util = [[LoadUtil alloc] initWithValues:appLevel nextLevel:nextLevel];


                switch (appLevel.type) {
                        case IMAGE_TEXT_DESCRIPTION_ACTIVITY:
                                loadSelector = @selector(loadImageTextNextLevel:);
                                break;
                        case IMAGE_LIST_ACTIVITY:
                                loadSelector = @selector(loadImageListNextLevel:);
                                break;
[…]


//call nextLevel of each view
      -(void) loadImageTextNextLevel:(LoadUtil*) util;
      -(void) loadImageListNextLevel:(LoadUtil*) util;
      -(void) loadListNextLevel:(LoadUtil*) util;
      -(void) loadImageZoomNextLevel:(LoadUtil*) util;
      -(void) loadImageGalleryNextLevel:(LoadUtil*) util;
      -(void) loadButtonsNextLevel:(LoadUtil*) util;
      -(void) loadFormNextLevel:(LoadUtil*) util;
      -(void) loadMapNextLevel:(LoadUtil*) util;
      -(void) loadVideoNextLevel:(LoadUtil*) util;
      -(void) loadWebNextLevel:(LoadUtil*) util;
      -(void) loadPdfNextLevel:(LoadUtil*) util;
      -(void) loadQRNextLevel:(LoadUtil*) util;
      -(void) loadCalendarNextLevel:(LoadUtil*) util;
      -(void) loadQuizNextLevel:(LoadUtil*) util;
      -(void) loadCanvasNextLevel:(LoadUtil*) util;
      -(void) loadAudioNextLevel:(LoadUtil*) util;

      -(void) loadSearchNextLevel;



/**
 * Load the data from calendar.xml into CalendarLevelData and also call NwCalendarController
 *
 * @param util
 *
 * @see loadCalendarController
 */
- (void) loadCalendarNextLevel:(LoadUtil*) util{
      NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

      //Read calendar.xml and save it into theData (CalendarLevelData type)
      CalendarLevelData* theData = [[NwUtil instance] readCalendarData:util.appLevel

nextLevel:util.nextLevel];

      //Charge data into controlerUtil
      LoadControlerUtil* controlerUtil = nil;
      controlerUtil = [[LoadControlerUtil alloc] initWithValues:util.appLevel

theData:theData];

      [util release];

      //loadCalendarController and we init its data with controlerUtil data
      [self performSelectorOnMainThread:@selector(loadCalendarController:)
                                      withObject:controlerUtil
                                   waitUntilDone:NO];

      [pool drain];
}



/**
 * Load NwCalendarController with all data ready to use
```

```
 * even asing nib file to controller
 *
 * @param theControlerUtil
 *
 * @see loadController
 */
- (void) loadCalendarController:(LoadControlerUtil*)  theControlerUtil{
      //Getting nib name

      NSString* controllerNibName = [eMobcViewController
addIPadSuffixWhenOnIPad:@"NwCalendarController"];
      //Init controller (NwCalendarController) with a specific nib name
      NwCalendarController *controller = [[NwCalendarController alloc]
initWithNibName:controllerNibName

                bundle:nil];
      controller.data = (CalendarLevelData*)theControlerUtil.data;
      controller.mainController = self;

      //Asing the NwController (currentController) the new controller of the new view
      self.currentController = controller;

      //load and show controller whit a transition
      [self loadController:currentController withAppLevel:theControlerUtil.appLevel];
      [controller release];
      [theControlerUtil release];
}


@interface NwUtil: NSObject {
//Objetos
      ApplicationData *theAppData;
      Cover* theCover;
      TopMenuData* theTopMenu;
      BottomMenuData* theBottomMenu;

      FormatsStylesLevelData* theFormatsStyles;
      StylesLevelData* theStyles;

      ProfileLevelData *theProfile;
}



//For Special Parsers
      -(ApplicationData *) readApplicationData;
      -(Cover*) readCover;
      -(FormatsStylesLevelData*) readFormats;
      -(StylesLevelData*) readStyles;
      -(ProfileLevelData*) readProfile;
      -(TopMenuData*) readTopMenu;
      -(BottomMenuData*) readBottomMenu;
      -(AppLevel*) readAppLevel:(NextLevel*)nextLevel;
      -(DataItem*) readAppLevelData:(NextLevel*)nextLevel;


/**
 * Read app.xml data
 *
 * @return read data
 */
-(ApplicationData *) readApplicationData {

      if (theAppData == nil) {
        NSString *path = [[[NSBundle mainBundle] resourcePath]
stringByAppendingPathComponent:@"app.xml"];

        AppParser* appParser = [[AppParser alloc] init];
        [appParser parseXMLFileAtURL:path];

        theAppData = appParser.currentApplicationDataObject;
        [appParser release];
      }
```

```
	return theAppData;
}



//For rest of Parsers
	-(ImageTextDescriptionLevelData*) readImageTextDescriptionData:(AppLevel*) appLevel nextLevel:
(NextLevel*)nextLevel;
	-(ImageListLevelData*) readImageListData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
	-(ListLevelData*) readListData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
	-(ImageZoomLevelData*) readImageZoomData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
	-(ImageGalleryLevelData*) readImageGalleryData:(LoadUtil*) util;
	-(ButtonsLevelData*) readButtonsData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
	-(FormLevelData*) readFormData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
	-(MapLevelData*) readMapData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
	-(VideoLevelData*) readVideoData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
	-(WebLevelData*) readWebData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
	-(PdfLevelData*) readPdfData:(AppLevel *)appLevel nextLevel:(NextLevel *)nextLevel;
	-(QRLevelData*) readQRData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel;
	-(CalendarLevelData*) readCalendarData:(AppLevel *)appLevel nextLevel:(NextLevel *)nextLevel;
	-(QuizLevelData*) readQuizData:(AppLevel *)appLevel nextLevel:(NextLevel *)nextLevel;
	-(CanvasLevelData*) readCanvasData:(AppLevel *)appLevel nextLevel:(NextLevel *)nextLevel;
	-(AudioLevelData*) readAudioData:(AppLevel *)appLevel nextLevel:(NextLevel *)nextLevel;


/**
 * Read calendar.xml
 *
 * @param appLevel Level View
 * @param nextLevel Next Level which this view can show
 *
 * @return Calendar data
 */
-(CalendarLevelData*) readCalendarData:(AppLevel*) appLevel nextLevel:(NextLevel*)nextLevel {
	CalendarParser *parser = [[CalendarParser alloc] init];

	NSData* parseData = [appLevel.file content];
	[parser parseXMLFileFromData:parseData];

	CalendarLevel* theLevel = (CalendarLevel*)parser.parsedLevel;

	DataItem* theItem = nil;
	if (nextLevel.dataId != nil) {
	  theItem = [theLevel dataItemById:nextLevel.dataId];
	}else {
	  theItem = [theLevel dataItemByNumber:nextLevel.dataNumber];
	}

	[parser release];
	return (CalendarLevelData*)theItem;
}


//Objetos
[…]
	eMobcViewController* mainController;
[…]



[esquema ]
el sideMenu ya no se utiliza por ahora y el NwButtonsController hereda de NwController.



//Publicity
	GADBannerView * admobBanner_; //admob
	SmartAdServerView *yocBanner_; //yoc


-(vid)viewDidLoad {
[…]
```

```
//publicity
      if([mainController.appData.banner isEqualToString:@"admob"]){
        [self createAdmobBanner];
      }else if([mainController.appData.banner isEqualToString:@"yoc"]){
        [self createYocBanner];
      }
[...]
}
```